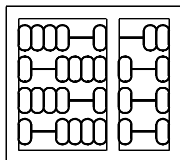
MC202: Estruturas de Dados

2º semestre de 2006

Tomasz Kowaltowski



Instituto de Computação – UNICAMP

`http://www.ic.unicamp.br/~mc202`

Parte do material contido nestas transparências foi copiada ou adaptada da apostila *Estruturas de Dados e Técnicas de Programação* de autoria de Cláudio L. Lucchesi e Tomasz Kowaltowski.

Pré-requisitos e objetivos

Pré-requisitos

Conteúdo de MC102, com programação em C

Objetivos

- Programação em (relativamente) baixo nível
- Técnicas de programação e estruturação de dados
- Preparação para
  - Análise de algoritmos
  - Programação de sistemas
  - Programação em geral
  - Bancos de dados
  - Engenharia de software

Programa da disciplina

- Introdução à análise de algoritmos
- Estruturação elementar de dados: matrizes, registros, apontadores
- Estruturas lineares: pilhas, filas, filas duplas
- Recursão e retrocesso
- Árvores binárias: representação, percursos
- Árvores gerais: representação, percursos
- Aplicação de árvores: busca, filas de prioridades, árvores *AVL*, árvores *B*
- Listas generalizadas
- Espalhamento

(continua)

- Assuntos complementares
  - Processamento de cadeias de caracteres
  - Gerenciamento de memória
  - Algoritmos de ordenação
  - Algoritmos em grafos
  - Tipos abstratos de dados e orientação a objetos

Avaliação

Provas (datas a serem confirmadas):

$$P = (2P_1 + 3P_2 + 5P_3)/10$$

- P_1: 21 de setembro
- P_2: 26 de outubro
- P_3: 28 de novembro

Laboratórios: (cerca de 12 tarefas)

$$L = \frac{\sum_{i=1}^{n} p_i t_i}{\sum_{i=1}^{n} p_i}$$

onde n é o número de tarefas de laboratório, p_i é o peso atribuído à tarefa ($0 \leq p_i \leq 1$) e t_i ($0 \leq t_i \leq 10$) é a nota obtida na tarefa.

Aproveitamento final:

$$A = \begin{cases} (6P + 4L)/10 & \text{se } P \geq 5 \text{ e } L \geq 5 \\ \min(P, L) & \text{caso contrário.} \end{cases}$$

A *média final* do curso F será calculada por:

$$F = \begin{cases} (A + E)/2 & \text{se o aluno fez o exame final} \\ A & \text{caso contrário} \end{cases}$$

onde E é a nota obtida no exame:

Exame final: 12 de dezembro

Observações:

- Será estabelecido um limite para o número de submissões de cada tarefa de laboratório.
- A submissão de uma tarefa de laboratório poderá ser considerada rejeitada se não seguir **estritamente** as exigências do enunciado, mesmo que produza resultados corretos nos testes.
- Qualquer tentativa de fraude nas provas ou nos laboratórios implicará em aproveitamento zero no semestre para todos os envolvidos, sem prejuízo de outras sanções.
- As transgressões às regras de uso dos sistemas computacionais implicarão em aproveitamento zero no semestre para todos os envolvidos, sem prejuízo de outras sanções.
- Não haverá provas substitutivas.
- Todas as provas e o exame final serão realizados sem consulta.

Material:

- Apostila
- Transparências

Disponíveis para cópia na gráfica do Departamento de Artes Cênicas (IA), em frente ao IC-1.

Introdução à Análise de Algoritmos

Importância da escolha de estrutura de dados – busca de um elemento numa seqüência:

(a)

```
...
x = a[k];
...
```

(b)

```
...
p = a;  i = 0;
while (i<k) {
  p = p−>prox;
  i++;
}
x = p−>info;
...
```

(a) Número de operações constante (vetor).

(b) Número de operações proporcional a k (lista ligada).

Exemplo de análise de trechos de programas

(a)

```
...
x = a+b;
...
```

(b)

```
...
for (i=0; i<n; i++)
    x = a+b;
...
```

(c)

```
...
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    x = a+b;
...
```

	a	*b*	*c*
análise simples (1)	1	n	n^2
análise detalhada (2)	2	$5n+2$	$5n^2+5n+2$

(1): atribuições (2): atribuições, operações aritméticas e comparações

As duas análises produzem resultados proporcionais para valores crescentes de n.

Noção intuitiva da notação O()

$$
\begin{aligned}
c &= \mathrm{O}(1) \qquad \text{para qualquer constante } c\\
2 &= \mathrm{O}(1)\\
5n+2 &= \mathrm{O}(n)\\
5n^2+5n+2 &= \mathrm{O}(n^2)\\
n^2 &= \mathrm{O}(n^3)\\
n^k &= \mathrm{O}(n^{k+1}), \qquad k \geq 0\\
\log_a n &= \mathrm{O}(\log_b n), \qquad a, b > 0\\
\log_2 n &= \mathrm{O}(\log_{10} n)
\end{aligned}
$$

Exemplo de análise de um procedimento de ordenação

```
void Ordena(int v[], int n) {
  int i, k, m, t;
  for (i=0; i<n−1; i++) {
    m = i;
    for (k=i+1; k<n; k++)
      if (v[k]<v[m])  m = k;
    t = v[i]; v[i] = v[m]; v[m] = t;
  }
} /* Ordena */
```


Número de comparações de elementos de v, para cada valor de i, é $n - i - 1$. O número total de comparações será:

$$\sum_{i=0}^{n-2}(n-i-1) = \frac{n^2}{2} + \frac{n}{2}$$

ou seja, o número de comparações é da ordem de $O(n^2)$.

Crescimento de algumas funções

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4.096	65.536
32	5	160	1.024	32.768	4.294.967.296
64	6	384	4.096	262.144	$\approx 18 \times 10^{18}$
128	7	896	16.384	2.097.152	$\approx 34 \times 10^{37}$

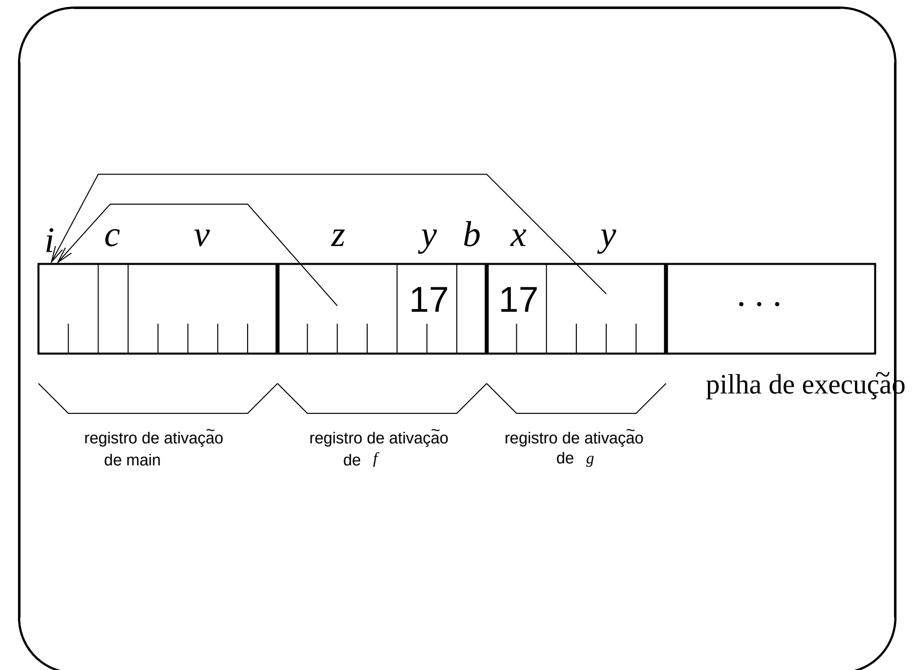
Execução de programas

Exemplo de funções simples

```
void g(int x, int *y) {
  *y = x;
} /* g */

void f(int *z) {
  int y;  char b;
  y = 17;
  g(y,z);
} /* f */
```

```
int main() {
  int i;  char c;
  char v[5];
  f(&i);
  return 0;
} /* main */
```

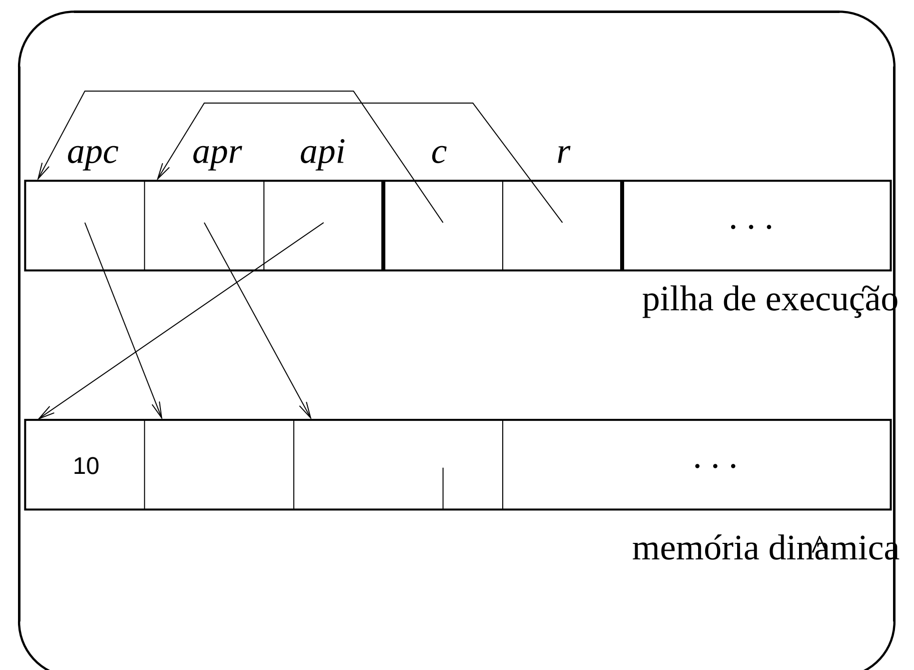
Exemplo de funções com alocação dinâmica

```
#include <malloc.h>
typedef char Cadeia[5];
typedef Cadeia *ApCadeia;
typedef struct {
  Cadeia nome;  int idade;
} Reg, *ApReg;

ApCadeia apc;
ApReg apr;
int *api;
```

```
void Aloca(ApCadeia *c, ApReg *r) {
  api = (int *)malloc(sizeof(int));
  *api = 10;
  *c = (ApCadeia)malloc(sizeof(Cadeia));
  *r = (ApReg)malloc(sizeof(Reg));
} /* Aloca */

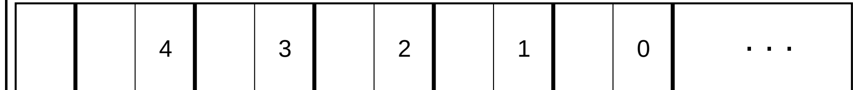
int main() {
  Aloca(&apc,&apr); free(apc);
  free(apr); free(api);
  return 0;
} /* main */
```

Exemplo de função recursiva

```
int fat(int n) {
 if (n==0)
  return 1;
 else
  return n*fat(n−1);
} /* fat */

int main() {
 int m;
 m = fat(4);
 return 0;
} /* main */
```

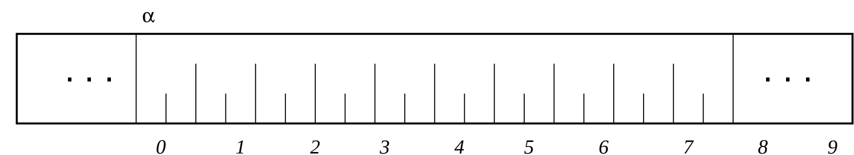



pilha de execução

Estruturas seqüenciais

Cálculo de endereços de vetores

Declaração: **int** a[10]; (ou a: **array** [0..9] **of** Integer;)



$$\text{ender}(a[i]) = \alpha + 2i$$

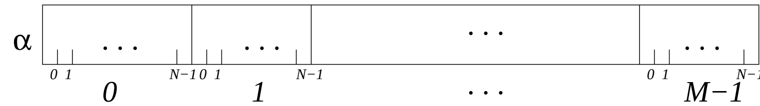
(Supondo inteiros de dois *bytes*.)

Número de operações: uma multiplicação e uma soma.

Cálculo de endereços de matrizes em geral

Declaração: T b[M][N]; (ou b: **array** [0..M−1,0..N−1] **of** T;)

$|T|$ é número de *bytes* de um elemento do tipo T.



$$\text{ender}(b[i,j]) = \alpha + |T|Ni + |T|j$$

Como $T_N = |T|N$ é uma constante, podemos reescrever:

$$\text{ender}(b[i,j]) = \alpha + T_N i + |T|j$$

Número de operações: duas multiplicações e duas somas. Este resultado pode ser estendido facilmente para matrizes de dimensões mais altas (exercício).

Formas especiais - exemplo de matriz triangular inferior

$$\begin{vmatrix} -10 & 0 & 0 & 0 & 0 & 0 \\ 8 & 20 & 0 & 0 & 0 & 0 \\ 0 & 5 & 7 & 0 & 0 & 0 \\ 25 & -3 & 1 & 0 & 0 & 0 \\ 5 & 10 & 8 & -4 & 10 & 0 \\ 10 & 8 & -3 & 1 & 9 & 25 \end{vmatrix}$$

Dos 36 elementos, 15 serão sempre nulos e 21 poderão ser não nulos. No caso geral, dos n^2 elementos, $(n-1)n/2$ serão sempre nulos e $n(n+1)/2$ poderão ser não nulos. Neste caso, pode-se usar uma representação linearizada ilustrada por:

$$\begin{array}{ccccccc} a[0,0] & a[1,0] & a[1,1] & \dots & a[5,0] & \dots & a[5,5] \\ \updownarrow & \updownarrow & \updownarrow & & \updownarrow & & \updownarrow \\ ra[0] & ra[1] & ra[2] & \dots & ra[15] & \dots & ra[20] \end{array}$$

O cálculo dos endereços fica para exercício.

Tarefa de laboratório 00

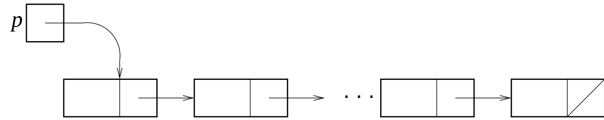
Treino; submissão recomendada.

Tarefa de laboratório 01

Um programa simples de manipulação de grafos.

Estruturas ligadas

Listas ligadas simples



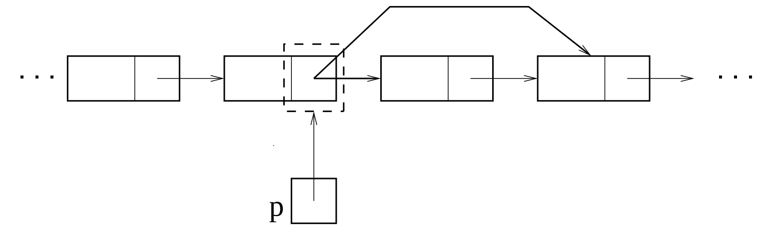
Declarações (equivalentes):

```
typedef
  struct RegLista *Lista;

typedef
  struct RegLista {
    T info;
    Lista prox;
  } RegLista;
```

```
typedef
  struct RegLista {
    T info;
    struct RegLista *prox;
  } RegLista, *Lista;
```


Inserção e remoção com passagem por referência



Inserção com passagem por referência

```
void InsereLista(Lista *p, T x) {

  Lista q = (Lista)malloc(sizeof(RegLista));

  q->info = x;
  q->prox = *p;
  *p = q;

} /* InsereLista */
```


Remoção com passagem por referência

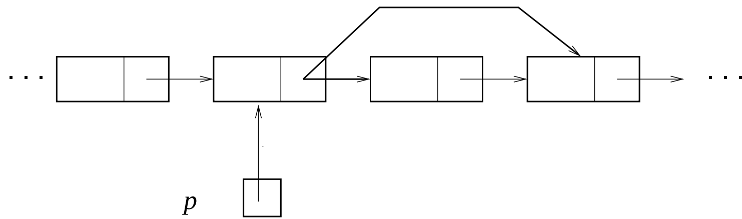
```
void RemoveLista(Lista *p, T *x) {

  Lista q = *p;

  *x = q->info;
  *p = q->prox;
  free(q);

} /*  RemoveLista */
```


Inserção e remoção com passagem por valor (apontador ao nó anterior)



Inserção com passagem por valor (apontador ao nó anterior)

```
void InsereLista(Lista p, T x) {

  Lista q = (Lista)malloc(sizeof(RegLista));

  q->info = x;
  q->prox = p->prox;
  p->prox = q;

} /* InsereLista */
```


Remoção com passagem por valor (apontador ao nó anterior)

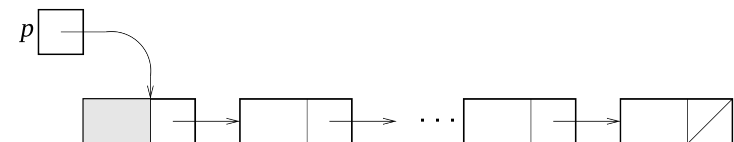
```
void RemoveLista(Lista p, T *x) {

  Lista q = p->prox;

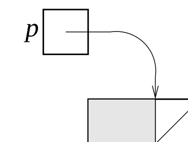
  *x = q->info;
  p->prox = q->prox;
  free(q);

} /*  RemoveLista */
```

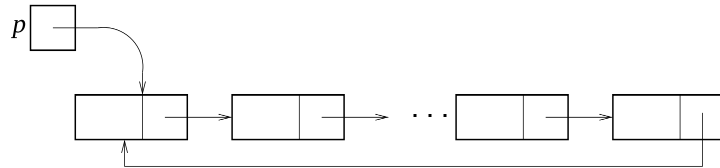

Lista simples com nó cabeça



Lista vazia:

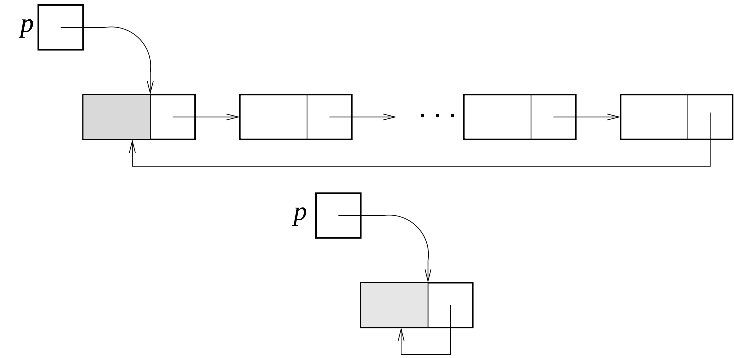


Lista simples circular



Problema: lista vazia

Lista circular com nó cabeça



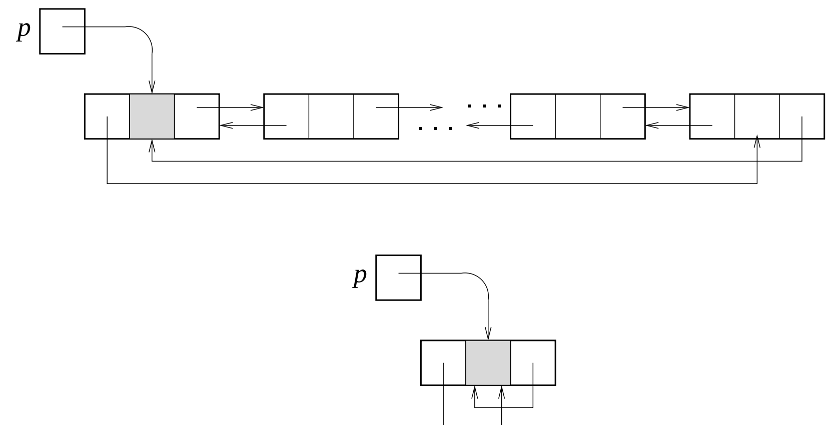
Busca em lista circular com nó cabeça – *sentinelas*

```
Lista BuscaCircular(Lista p, T x) {
 /* Busca sem sentinela */
  Lista q = p;

  do {
    q = q−>prox;
  } while ((q!=p) && (q−>info!=x));
  if (q==p)
    return NULL;
  else
    return q;
} /* BuscaCircular */
```

```
Lista BuscaCircular(Lista p, T x) {
/* Busca com sentinela */
  Lista q = p;
  q−>info = x;
  do {
    q = q−>prox;
  } while (q−>info!=x);
  if (q==p)
    return NULL;
  else
    return q;
} /* BuscaCircular */
```


Lista duplamente ligada com nó cabeça



Operações sobre lista duplamente ligada

```
typedef
  struct RegListaDupla {
    T info;
    struct RegListaDupla *esq,*dir;
  } RegListaDupla, *ListaDupla;
```

```
void InsereDuplaEsq(ListaDupla p, T x) {
  ListaDupla q = (ListaDupla)malloc(
          sizeof(RegListaDupla));
  q->info = x;
  q->esq = p->esq;
  q->dir = p;
  p->esq->dir = q;
  p->esq = q;
} /* InsereDuplaEsq */
```

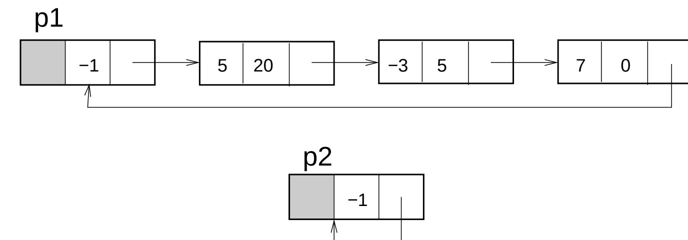
```
void RemoveDupla(
        ListaDupla p, T *x) {
  /* Remove p; supõe mais de um
     nó na lista */
  p->esq->dir = p->dir;
  p->dir->esq = p->esq;
  *x = p->info;
  free(p);
} /* RemoveDupla */
```


Exemplo: operações com polinômios (parecido com tarefa de laboratório 02)

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0 x^0$$

onde $a_n \neq 0$, exceto possivelmente no caso $n = 0$.

Representação ligada de $P_1(x) = 5x^{20} - 3x^5 + 7$ e $P_2(x) = 0$:



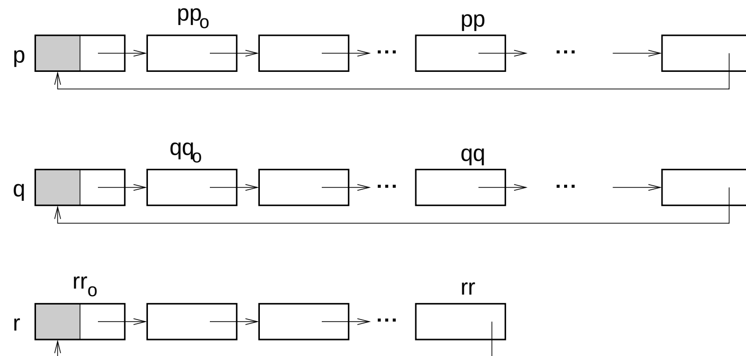
Declarações para polinômios

```
typedef struct AuxPol {
  int   expo;
  float coef;
  struct AuxPol *prox;
} Termo, *Polinomio;
```


Exemplo de rotina: impressão (C)

```
void ImprimePolinomio(Polinomio p) {
  if (p->prox==p) {
    printf("Polinômio nulo\n");
    return;
  }
  p = p->prox;
  while (p->expo!=-1) {
    printf("(%2d,%5.1f) ",p->expo,p->coef);
    p = p->prox;
  }
  printf("\n");
} /* ImprimePolinomio */
```

Soma de polinômios: paradigma de intercalação

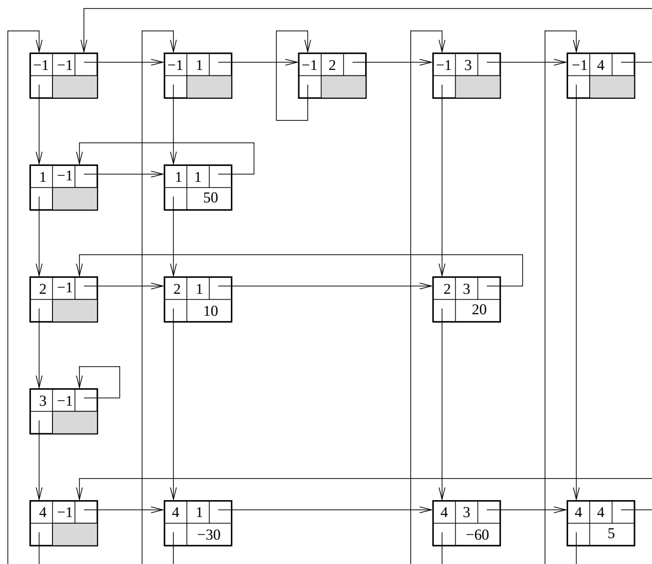


pp_0, qq_0 e rr_0: valores iniciais de pp, qq e rr

Exemplo de matriz esparsa

$$\begin{vmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{vmatrix}$$

Implementação: listas ortogonais



Operações sobre matrizes esparsas

```
typedef
  struct RegEsparsa {
    int linha, coluna;
    double valor;
    struct RegEsparsa *direita, *abaixo;
  } RegEsparsa, *Matriz;

void InincializaMatriz(Matriz *a, int m, int n);
void LiberaMatriz(Matriz *a);
double ElementoMatriz(Matriz *a, int i, int j);
void AtribuiMatriz(Matriz *a, int i, int j, double x);
void SomaMatrizes(Matriz *a, Matriz *b, Matriz *c);
void MultiplicaMatrizes(Matriz *a, Matriz *b, Matriz *c);
```

Estruturas lineares

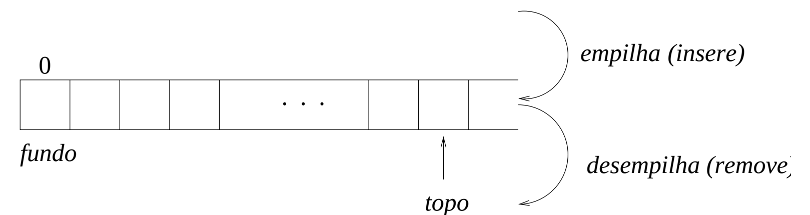
Estruturas lineares em geral: operações típicas

- selecionar e modificar o k-ésimo elemento;
- inserir um novo elemento entre as posições k e $k+1$;
- remover o k-ésimo elemento;
- concatenar duas seqüências;
- desdobrar uma seqüência;
- copiar uma seqüência;
- determinar o tamanho de uma seqüência;
- buscar um elemento que satisfaz uma propriedade;
- ordenar uma seqüência;
- aplicar um procedimento a todos os elementos de uma seqüência;
- ...

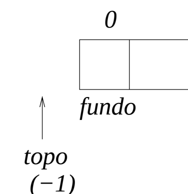
Estruturas lineares particulares

- Pilha (*stack*): inserção e remoção na mesma extremidade da estrutura
- Fila (*queue*): inserção numa extremidade (fim) e remoção na outra extremidade (início)
- Fila dupla (*double ended queue*): inserção e remoção em ambas extremidades da estrutura

Pilha: implementação seqüencial



Pilha vazia:



Pilha: implementação ligada

pilha (topo)

fundo

Aplicações de pilhas

- Processamento de linguagens parentéticas:
  - linguagens de programação
  - XML
- Implementação da recursão
- Avaliação expressões em notação pós-fixa (notação polonesa reversa)

Exemplo de aplicação simples: balanceamento de parênteses

Correto	Incorreto
ϵ	(
()	)
[()]	[)
[]()[()[]]	()()[
((([[[]]])))	)(

Pilha	*Resto da seqüência*
Vazia	([([][()])])
(	[([][()])])
([	([][()])])
([(	[][()])])
([([	][()])])
([(	[()])])
([([	()])])
([([(	)])])
([([	])])
([(	)])
([	])
(	)
Vazia	ϵ

Implementação ligada de pilhas

```
typedef struct ElemPilha {
  T info;
  struct  ElemPilha *prox;
} ElemPilha, *Pilha;
```

```
void Empilha(Pilha *p, T x) {
  Pilha q = (Pilha)malloc(sizeof(ElemPilha));
  q−>info = x;
  q−Prox = *p;
  *p = q;
} /* Empilha */
```


Implementação seqüencial de pilhas

```
typedef
  struct {
    int topo;
    T elementos[TAM_MAX];
  } Pilha;
```

```
void Empilha(Pilha *p, T x) {
  if ((*p).topo==(TAM_MAX−1))
    TrataErro("Pilha cheia");
  (*p).topo++;
  ((*p).elementos)[(*p).topo] = x;
} /* Empilha */
```


Notações para expressões aritméticas

infixa	*pós-fixa*	*pré-fixa*
a	a	a
$a+b$	$ab+$	$+ab$
$a+b*c$	$abc*+$	$+a*bc$
$(a+b)*c$	$ab+c*$	$*+abc$

Exemplo: avaliação de expressões sob forma pós-fixa

Notação pós-fixa: $(3+5)*2-(10-3)/2 \quad\Longrightarrow\quad 3\ \ 5+2\star 10\ \ 3-2/-$

Estados consecutivos da pilha:

Vazia	$3\ \ 5+2\star 10\ \ 3-2/-$
3	$5+2\star 10\ \ 3-2/-$
3 5	$+2\star 10\ \ 3-2/-$
8	$2\star 10\ \ 3-2/-$
8 2	$\star 10\ \ 3-2/-$
16	$10\ \ 3-2/-$
16 10	$3-2/-$
16 10 3	$-2/-$
16 7	$2/-$
16 7 2	$/-$
16 3	$-$
13	Vazia

Exemplo: transformação de notação infixa para pós-fixa

$a * b + c * d \wedge e / f - g * h$

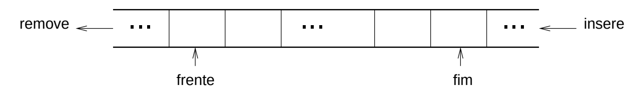
$a\ b * c\ d\ e \wedge * f / + g\ h * -$

Saída	*Pilha*	*Entrada*
		$a * b + c * d \wedge e/f - g * h$
a		$* b + c * d \wedge e/f - g * h$
a	$*$	$b + c * d \wedge e/f - g * h$
ab	$*$	$+ c * d \wedge e/f - g * h$
$ab*$		$+ c * d \wedge e/f - g * h$
$ab*$	$+$	$c * d \wedge e/f - g * h$
$ab * c$	$+$	$*d \wedge e/f - g * h$
$ab * c$	$+*$	$d \wedge e/f - g * h$
$ab * cd$	$+*$	$\wedge e/f - g * h$
$ab * cd$	$+ * \wedge$	$e/f - g * h$
$ab * cde$	$+ * \wedge$	$/f - g * h$
$ab * cde\wedge$	$+*$	$/f - g * h$

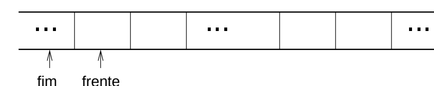
(continua)

Saída	*Pilha*	*Entrada*
$ab * cde \wedge *$	$+$	$/f - g * h$
$ab * cde \wedge *$	$+/$	$f - g * h$
$ab * cde \wedge *f$	$+/$	$- g * h$
$ab * cde \wedge *f/$	$+$	$- g * h$
$ab * cde \wedge *f/+$		$- g * h$
$ab * cde \wedge *f/+$	$-$	$g * h$
$ab * cde \wedge *f/ + g$	$-$	$* h$
$ab * cde \wedge *f/ + g$	$-*$	h
$ab * cde \wedge *f/ + gh$	$-*$	
$ab * cde \wedge *f/ + gh*$	$-$	
$ab * cde \wedge *f/ + gh * -$		

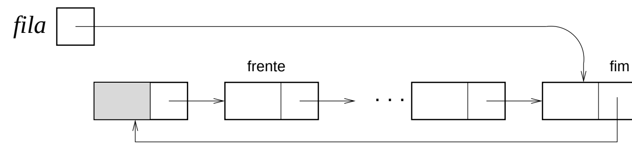
Fila: implementação seqüencial



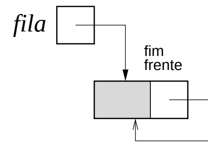
Fila vazia:



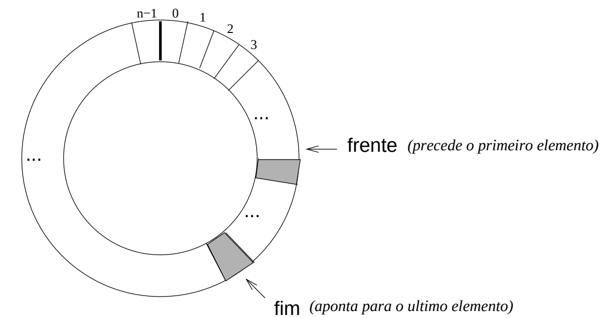
Fila: implementação ligada circular



Fila vazia:



Fila: implementação seqüencial circular



Condições:

- Inicial: $frente = fim = 0$
- Fila vazia: $frente = fim$
- Fila cheia: $frente = fim$ (solução: $frente = (fim + 1)\%n$)

Implementação seqüencial circular de filas

```
#define TAM_MAX_FILA 1000

typedef struct {
  int frente, fim;
  T elementos[TAM_MAX_FILA];
} Fila;

void InsereFila(Fila *f, T x) {
  if ((*f).frente==(((*f).fim+1)%TAM_MAX_FILA))
    TrataErro("Fila cheia");
  (*f).fim = ((*f).fim+1)%TAM_MAX_FILA;
  (*f).elementos[(*f).fim] = x;
} /* InsereFila */
```


Recursão e retrocesso

(*backtracking*)

Recursão – Exemplo 1: fatorial

```
int fatorial(int n) {
  if (n==0)
    return 1;
  else
    return n*fatorial(n−1);
} /* fatorial */
```

```
int fatorial(int n) {
  int i,f=1;
  for (i=1; i<=n; i++)
    f = f*i;
  return f;
} /* fatorial */
```

Eficiência: ambos $O(n)$

Recursão – Exemplo 2: fibonacci

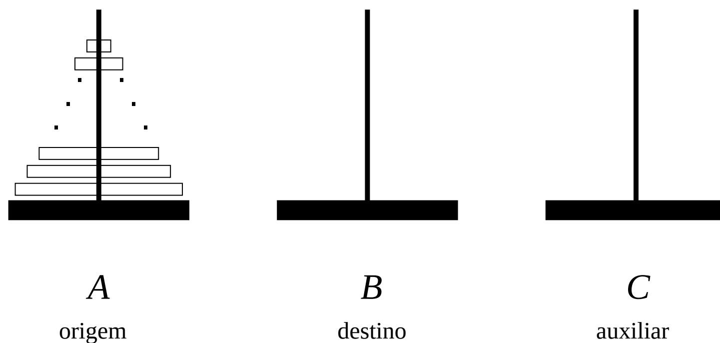
Seqüência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

```
int fibonacci(int n) {
  if (n<=1)
    return n;
  else
    return fibonacci(n−1)+fibonacci(n−2);
} /* fibonacci */
```

```
int fibonacci(int n) {
  int f1=0, f2=1, f3, i;
  for (i=1; i<=n; i++) {
    f3 = f1+f2;
    f1 = f2;  f2 = f3;
  }
  return f1;
} /* fibonacci */
```

Eficiência:	$O(1.6^n)$	$O(n)$
$n = 100$:	$\approx 10^{20}$ somas	100 somas

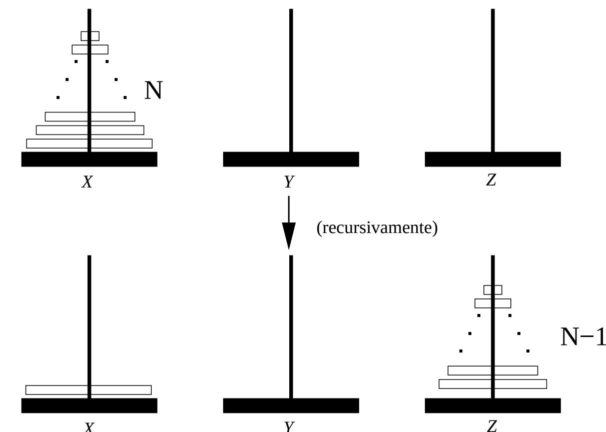
Recursão – Exemplo 3: Torres de Hanoi



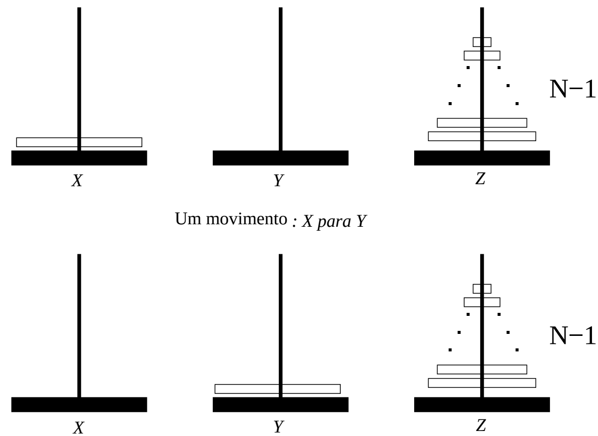
Regras:

- um disco de cada vez
- disco de diâmetro maior não pode ficar em cima de um de diâmetro menor

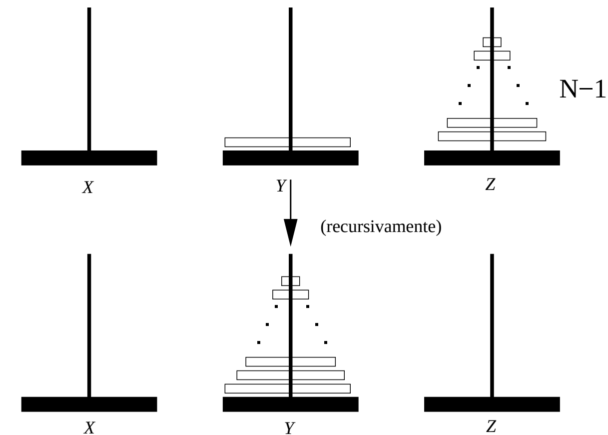
Passo 1: transferência recursiva de $N-1$ discos



Passo 2: movimento do maior disco



Passo 1: transferência recursiva final de $N-1$ discos



Programa *Hanoi*

```
void Hanoi(char org, char dest, char aux, int n) {
  if (n>0) {
    Hanoi(org, aux,dest,n−1);
    printf("Mova de %c para %c\n",org, dest);
    Hanoi(aux, dest,org,n−1);
  }
} /* Hanoi */
```

Chamada inicial:

Hanoi('A','B','C',64)

Vai demorar muito: $2^N - 1$ movimentos!

Exemplos de saída

```
N=1:

Mova de A para B


N=2:

Mova de A para C
Mova de A para B
Mova de C para B
```

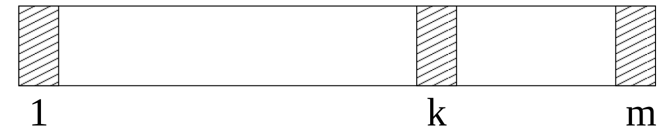
```
N=3:

Mova de A para B
Mova de A para C
Mova de B para C
Mova de A para B
Mova de C para A
Mova de C para B
Mova de A para B
```

Exemplo de saída para $N = 4$

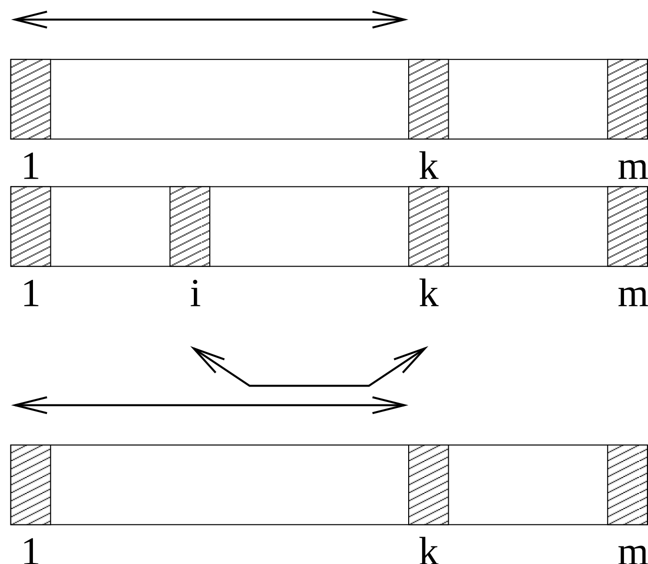
```
Mova de A para C     Mova de C para B
Mova de A para B     Mova de C para A
Mova de C para B     Mova de B para A
Mova de A para C     Mova de C para B
Mova de B para A     Mova de A para C
Mova de B para C     Mova de A para B
Mova de A para C     Mova de C para B
Mova de A para B
```


Recursão – Exemplo 4: Geração de permutações



Gerar todas as permutações dos m elementos do vetor.

Suponha uma função $Permuta(k, m)$ que gera (imprime) todas as permutações dos elementos de 1 a k, seguidas dos elementos de $k + 1$ a m:



Restaura a situação anterior.

Função *Permuta*

```
void Permuta(int k, int m) {
 if (k==0)
  Imprime(m);
 else {
  int i;
  for (i=k; i>=0; i−−) {
   Troca(i,k);
   Permuta(k−1,m);
   Troca(i,k);
  }
 }
} /* Permuta */
```


Saída de *Permuta(3,3)*

```
1  2  3
2  1  3
1  3  2
3  1  2
3  2  1
2  3  1
```

Desafio: imprimir em ordem lexicográfica.

Exemplo de retrocesso: movimentos do cavalo

Movimentos possíveis no jogo de xadrez:

	2		1	
3				0
		●		
4				7
	5		6	

Exemplo de retrocesso: movimentos do cavalo (cont)

Um percurso da posição (0,0) até (4,4):

	0	1	2	3	4
0	1	4	9	12	
1	10	13	6	3	
2	5	2	11	8	
3		7	14		
4					15

Soluções

1. Acha uma solução
2. Acha uma solução que cobre todas as posições livres
3. Enumera todas as soluções

Observação: Esta não é a melhor maneira de resolver este problema mas ilustra bem o mecanismo geral de retrocesso.

Inicialização

```
#include <stdio.h>
#define TAM_MAX 20
#define NUM_MOV 8
typedef enum {false, true} Boolean;
int tab[TAM_MAX][TAM_MAX];
int dl[NUM_MOV] = { −1, −2, −2, −1, 1, 2, 2, 1 };
int dc[NUM_MOV] = { 2, 1, −1, −2, −2, −1, 1, 2 };

void ImprimeTab(int tam) {
  int i,j;
  for (i=0; i<tam; i++) {
    for (j=0; j<tam; j++)
      printf("%5d",tab[i][j]);
    printf("\n");
  }
} /* ImprimeTabuleiro */
```



```
Boolean TentaMovimento(int tam, int num, int lin, int col, int ld, int cd) {
  int k, lp, cp;
  Boolean res = false;
  if ((0<=lin) && (lin<tam) && (0<=col) && (col<tam) && (tab[lin][col]==0)) {
    tab[lin][col] = num;                /* tenta próximo movimento */
    if ((lin==ld) && (col==cd)) {       /* alcançou o destino */
      res = true; ImprimeTab(tam);
    } else { k = 0;             /* tenta prosseguir com um dos 8 movimentos */
          do {
            lp = lin+dl[k];  cp = col+dc[k];
            res = TentaMovimento(tam,num+1,lp,cp,ld,cd); k++;
          } while ((!res) && (k<NUM_MOV));
    }                           /* esgotou os movimentos ou achou */
    tab[lin][col] = 0;          /* desfaz a tentativa */
  }
  return res;
} /* TentaMovimento: acha uma solução */
```



```
Boolean TentaMovimento(int tam, int num, int lin, int col, int ld, int cd, int nocup) {
  int k, lp, cp;
  Boolean res = false;
  if ((0<=lin) && (lin<tam) && (0<=col) && (col<tam) && (tab[lin][col]==0)) {
    tab[lin][col] = num;                        /* tenta próximo movimento */
    if ((lin==ld) && (col==cd) && ((nocup+num)==tam*tam)) {
      res = true; ImprimeTab(tam);              /* alcançou o destino */
    } else { k = 0;             /* tenta prosseguir com um dos 8 movimentos */
          do {
            lp = lin+dl[k];  cp = col+dc[k];
            res = TentaMovimento(tam,num+1,lp,cp,ld,cd,nocup); k++;
          } while ((!res) && (k<NUM_MOV));
    }                           /* esgotou os movimentos ou achou */
    tab[lin][col] = 0;          /* desfaz a tentativa */
  }
  return res;
} /* TentaMovimento: cobre todas as posições */
```



```
void TentaMovimento(int tam, int num, int lin, int col, int ld, int cd) {
  int k, lp, cp;
  if ((0<=lin) && (lin<tam) && (0<=col) && (col<tam) && (tab[lin][col]==0)) {
    tab[lin][col] = num;                /* tenta próximo movimento */
    if ((lin==ld) && (col==cd)) {       /* alcançou o destino */
      ImprimeTab(tam);
    } else { k = 0;             /* tenta prosseguir com um dos 8 movimentos */
          do {
            lp = lin+dl[k];  cp = col+dc[k];
            TentaMovimento(tam,num+1,lp,cp,ld,cd); k++;
          } while (k<NUM_MOV);
    }                           /* esgotou os movimentos ou achou */
    tab[lin][col] = 0;          /* desfaz a tentativa */
  }
} /* TentaMovimento: achar todas as soluções */
```

Exemplo de entrada e saída (acha uma solução)

```
Entrada                Saída
---------------------------------

  5           1    4    9   12   -1
  0 0        10   13    6    3    0
  4 4         5    2   11    8    0
  0 4         0    7   14    0   -1
  3 4        -1    0    0    0   15
  4 0
 -1 -1
```


Exemplo de retrocesso: distância de edição

```
RECURSÃO E RETROCESSO

RCURÇÃO ER RTROCESOS
 ^  ^    ^  ^    ^ ^
 |  |    |  |    | |
 I  S    R  I    I R
```

Operações elementares:

- **A:** avanço (subentendido)
- **I:** inserção
- **S:** substituição
- **R:** remoção


```
int Distancia(char *teste, char *correta) {
  int dIns, dRem, dSub;       /* custos para cada operação */
  if (((*teste)==NUL_CHAR) && ((*correta)==NUL_CHAR))  /* ambas vazias */
     return 0;
  dIns = dRem = dSub = INT_MAX;
  if (((*teste)!=NUL_CHAR) && ((*correta)!=NUL_CHAR) &&
     ((*teste)==(*correta)))
    return Distancia(teste+1,correta+1);
  if (((*teste)!=NUL_CHAR) && ((*correta)!=NUL_CHAR)) /* ambas não vazias */
     dSub = custoSub+Distancia(teste+1,correta+1);
  if ((*teste)!=NUL_CHAR)
     dRem = custoRem+Distancia(teste+1,correta);
  if ((*correta)!=NUL_CHAR)
     dIns = custoIns+Distancia(teste,correta+1);
  return min(dIns, min(dRem, dSub));
} /* Distancia */
```


Distância de edição: desafios

- Melhorar o desempenho do algoritmo: o algoritmo é exponencial não sendo viável, sob esta forma, em aplicações práticas
- Imprimir o número de operações de cada tipo (avanço, inserção, remoção e substituição) para obter a solução
- Imprimir a seqüência de operações para obter a solução

Eliminação da recursão

Exemplo de esquema de função recursiva

```
void Exemplo(T1 x1, T2 x2, ...) {
  S1 y1; S2 y2; ...;
  Ci;         /∗ Comandos iniciais ∗/
  if (E(...)) {
    C0;       /∗ Caso base ∗/
  } else {    /∗ Chamadas recursivas ∗/
    C1;  Exemplo(e11,e12,...);
    C2;  Exemplo(e21,e22,...);
    C3;  Exemplo(e31,e32,...);
    ...;
    Cm;  Exemplo(em1,em2,...);
    Cf;
  }
} /∗ Exemplo ∗/
```


Eliminação da recursão - esquema

```
typedef enum {chamada1, chamada2, chamada3, ...} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;

void Exemplo(T1 x1, T2 x2, ...) {
  S1 y1; S2 y2; ...;    /∗ variáveis locais originais ∗/
  T1 t1, T2 t2, ...;    /∗ variáveis temporárias ∗/
  Pilha f; Chamadas ch; Acoes acao;  /∗ variáveis auxiliares ∗/
  InicializaPilha(&f); acao = entrada;
  do {
    switch (acao) {
      case (entrada): ... break;
      case (retorno): ... break;
      case (saida):  break;
    }
  } while (acao!=saida);
} /∗Exemplo∗/
```



```
case (entrada):
  Ci;        /∗ Comandos iniciais ∗/
  if (E(...)) {
    C0;  acao = retorno;     /∗ Caso base ∗/
  } else {    /∗ Primeira chamada recursiva ∗/
    C1; Empilha(f,x1,x2,...,y1,y2,...,chamada1);
    t1 = e11; t2 = e12; ...;
    x1 = t1;  x2 = t2;  ...;  /∗ Recalcula argumentos ∗/
  }
  break;
```

```
case (retorno):
 if (PilhaVazia(f))  acao = saida;
 else {
  Desempilha(f,&x1,&x2,...,&y1,&y2,...,&ch);
  switch (ch) {
   case (chamada1):
    C2; Empilha(f,x1,x2,...,y1,y2,...,chamada2);
    t1 = e21; t2 = e22; ...; x1 = t1;  x2 = t2;  ...; acao = entrada; break;
   case (chamada2):
    C3; Empilha(f,x1,x2,...,y1,y2,...,chamada3);
    t1 = e31; t2 = e32; ...; x1 = t1;  x2 = t2;  ...; acao = entrada; break;
   ...;
   case (chamadam):
    Cf; break;
  } /* switch (ch) */
 }
 break;
```


Exemplo: função fatorial

```
int fatorial(int n) {
 if (n==0)
  return 1;
 else
  return n*fatorial(n−1);
} /* fatorial */
```



```
typedef enum {chamada1} Chamadas;
typedef enum {entrada, saida, retorno} Acoes;

int fatorial(int n) {
 int res, t1;
 Pilha f; Chamadas ch; Acoes acao;
 InicializaPilha(&f); acao = entrada;
 do {
  switch (acao) {
   case (entrada): ... break;
   case (retorno): ... break;
   case (saida):  break;
  }
 } while (acao!=saida);
 return res;
} /* fatorial */
```



```
case (entrada):
 if (n==0) {
  res = 1;  acao = retorno;
 } else {
  Empilha(f,n,chamada1);
  t1 = n; n = t1−1;
 }
 break;
```

```
    case (retorno):
     if (PilhaVazia(f)) acao = saida;
     else {
      Desempilha(f,&n,&ch);
      switch (ch) {
       case (chamada1):
        res = n*res;
        break;
      }
     break;
```

Obs.: Note como neste caso a variável *res* está sendo usada para guardar o resultado da função.

Exemplo: função *Hanoi*

```
void Hanoi(char org, char dest, char aux, int n) {
  if (!(n>0))
   ;
  else {
    Hanoi(org, aux,dest,n−1);
    printf("Mova de %c para %c\n",org, dest);
    Hanoi(aux, dest,org,n−1);
  }
} /* Hanoi */
```



```
typedef enum {chamada1, chamada2};
typedef enum {entrada, saida, retorno} Acoes;
void Hanoi(char org, char dest, char aux, int n) {
 char t1; char t2; char t3; int t4;
 Pilha f; Chamadas ch; Acoes acao;
 InicializaPilha(&f); acao = entrada;
 do {
  switch (acao) {
   case (entrada): ...; break;
   case (retorno): ...; break;
   case (saida): break;
  }
 } } while (acao!=saida);
} /* Hanoi */
```



```
    case (entrada):
     if (!(n>0)) {
      acao = retorno;
     } else {
      Empilha(f,org,dest,aux,n,chamada1);
      t1 = org; t2 = aux; t3 = dest; t4 = n−1;
      org = t1; dest = t2; aux = t3; n = t4;
     }
     break;
```

```
case (retorno):
 if (PilhaVazia(f))
  acao = saida;
 else {
  Desempilha(f,&org,&dest,&aux,&n,&ch);
  switch (ch) {
   case (chamada1):
    printf("Mova de %c para %c\n",org, dest);
    Empilha(f,org,dest,aux,n,chamada2);
    t1 = aux; t2 = dest; t3 = org; t4 = n−1;
    org = t1; dest = t2; aux = t3; n = t4;
    acao = entrada;
    break;
   case (chamada2):
    break;
  }
  break;
```


Recursão mutua:

Análise sintática

Exemplo simples de recursão mútua

```
int g(int n);
int f(int n) {
 if (n==0)
  return 0;
 else
  return g(n−1);
} /* f */
int g(int n) {
 if (n==0)
  return 1;
 else
  return f(n−1);
} /* g */
```


Análise de expressões

Expressões com operadores binários ‘+’, ‘−’, ‘∗’, ‘/’ e os parênteses ‘(’ e ‘)’:

$$e = t_1 \oplus t_2 \oplus \cdots \oplus t_n, \qquad n \geq 1$$

$$t = f_1 \otimes f_2 \otimes \cdots \otimes f_n, \qquad n \geq 1$$

$$f = x \qquad \text{ou} \qquad f = (\, e \,)$$

Programa de tradução de infixa para pós-fixa:

```
char entrada[TAM_MAX];
char *pe;

void Expressao();
void Termo();
void Fator();

void InPos() {
  pe = &entrada[0];
  Expressao();
  if ((*pe)!='\0')
    Erro();
} /* InPos */
```


Fator

```
void Fator() {
  char corrente = *pe;
  switch (corrente) {
    case 'a': case 'b': ...: case 'z':
      Sai(corrente); pe++; break;
    case '(':
      pe++;
      Expressao();
      if ((*pe)==')')
        pe++;
      else
        Erro();
      break;
    default:
      Erro();
  }
} /* Fator */
```


Termo

```
void Termo() {
  char op;
  Fator();
  do {
    op = *pe;
    if ((op=='*')||(op=='/')) {
      pe++;
      Fator();
      Sai(op);
    } else
      break; /* do */
  } while (true);
} /* Termo */
```


Expressão

```
void Expressao() {
  char op;
  Termo();
  do {
    op = *pe;
    if ((op=='+')||(op=='-')) {
      pe++;
      Termo();
      Sai(op);
    } else
      break; /* do */
  } while (true);
} /* Expressao */
```

Operador de exponenciação

Fator redefinido:

$$f = p_1 \wedge p_2 \wedge \cdots \wedge p_n, \qquad n \geq 1$$

Primário:

$$p = x \qquad \text{ou} \qquad p = (\, e \,)$$

Prioridade? Solução:

$$f = p \qquad \text{ou} \qquad f = p \wedge f$$

Fator redefinido

```
void Fator() {
 Primario();
 if ((*pe)=='^') {
  pe++;
  Fator();
  Sai('^');
 }
} /* Fator */
```


Primário

```
void Primario() {
 corrente = *pe;
 switch (corrente) {
  case 'a': case 'b': ...: case 'z':
   Sai(corrente); pe++; break;
  case '(':
   pe++;
   Expressao();
   if ((*pe)==')')
    pe++;
   else
    Erro();
   break;
  default:
   Erro();
 }
} /* Primario */
```


Analogia para expressões e termos

$$e = t \qquad \text{ou} \qquad e = e \oplus t$$

$$t = f \qquad \text{ou} \qquad t = t \otimes f$$

Problemas:

como distinguir as alternativas

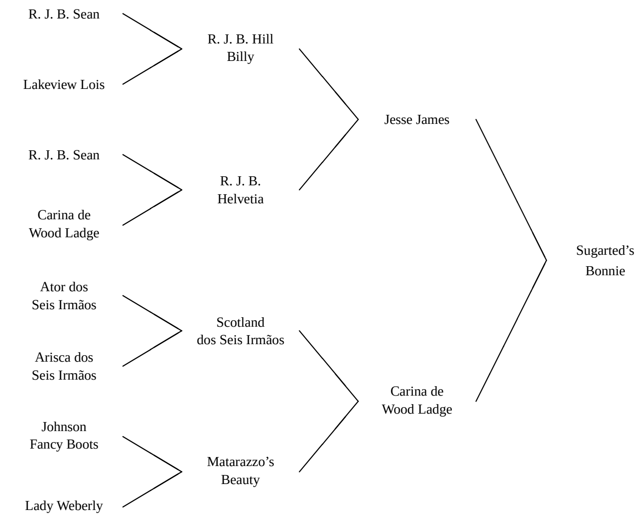
repetição infinita no segundo caso (recursão esquerda)

```
void Expressao() {
 ...;
 if (???)
  Termo();
 else
  Expressao();
 ...

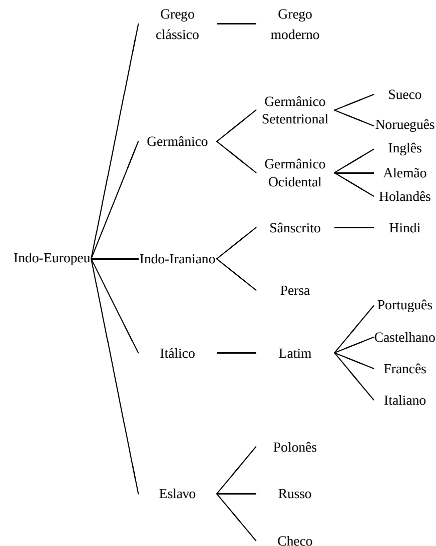
} /* Expressao */
```

Árvores binárias

Exemplo de árvore binária: árvore genealógica (*pedigree*)



Exemplo de árvore não binária: árvore de descendentes

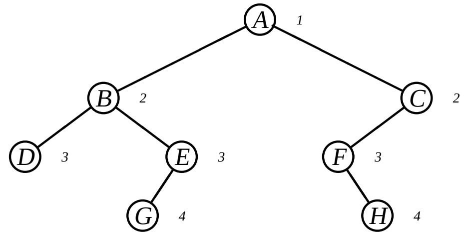


Definição

Uma *árvore binária* é um conjunto de *nós* que:

- ou é vazio (*árvore binária vazia*)
- ou contém um nó especial denominado *raiz da árvore* e o resto do conjunto está particionado em duas árvores binárias disjuntas (possivelmente vazias), denominadas *subárvore esquerda* e *subárvore direita*.

Conceitos



Raiz da árvore: A

Filho esquerdo de A: B

Pai de F: C

Descendentes de B: B, D, E e G

Folhas: D, G e H

Níveis: indicados

Subárvores binárias vazias: 9

Filho direito de A: C

Irmão de E: D

Antepassados de H: H, F, C e A

Nós internos: todos exceto as folhas

Altura (profundidade) – nível máximo: 4

Subárvores binárias não vazias: 7

Fatos sobre árvores binárias

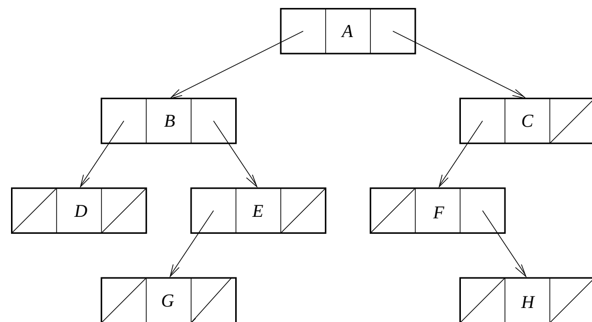
Uma árvore binária com n nós tem:

- altura máxima n
- altura mínima $\lceil \log_2(n+1) \rceil$
- subárvores vazias: $n+1$
- subárvores não vazias: $n-1$ (se $n>0$)

Uma árvore binária de altura h tem:

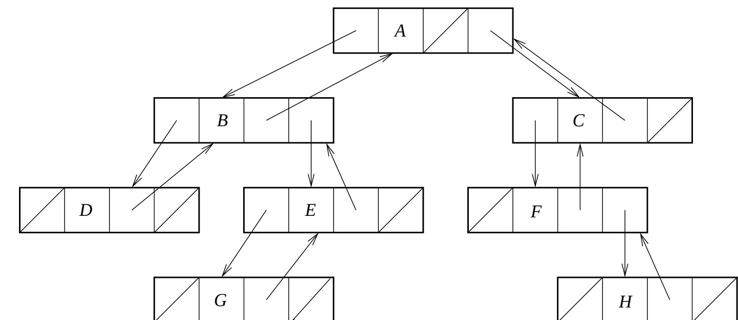
- no mínimo h nós
- no máximo $2^h - 1$ nós

Representação ligada comum

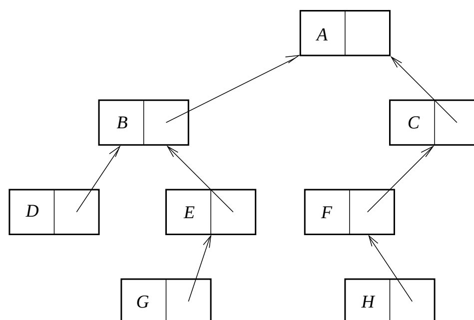


Para ter acesso a todos os nós basta ter um apontador à raiz.

Representação ligada com três apontadores



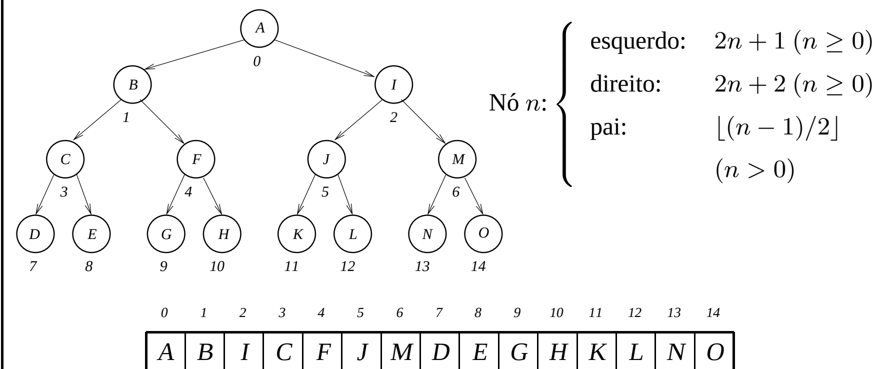
Representação com o campo *pai* apenas



Problemas:

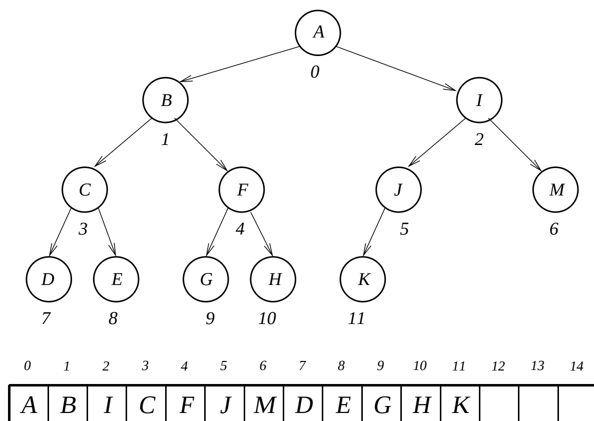
- precisa ter acesso a todas as folhas, pelo menos.
- não consegue distinguir entre os filhos esquerdos e direitos

Representação seqüencial: árvores binárias completas



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	I	C	F	J	M	D	E	G	H	K	L	N	O

Representação seqüencial: árvores binárias quase completas



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	I	C	F	J	M	D	E	G	H	K			

Percursos em profundidade

- *Pré-ordem:*

  Visitar a raiz
  Percorrer a subárvore esquerda em pré-ordem
  Percorrer a subárvore direita em pré-ordem

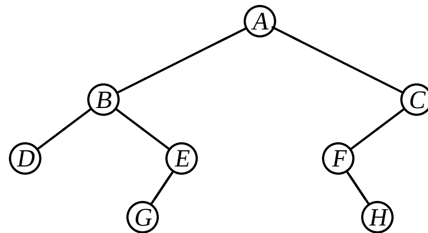
- *Pós-ordem:*

  Percorrer a subárvore esquerda em pós-ordem
  Percorrer a subárvore direita em pós-ordem
  Visitar a raiz

- *Inordem (ou ordem simétrica):*

  Percorrer a subárvore esquerda em inordem
  Visitar a raiz
  Percorrer a subárvore direita em inordem

Exemplos de percurso em profundidade



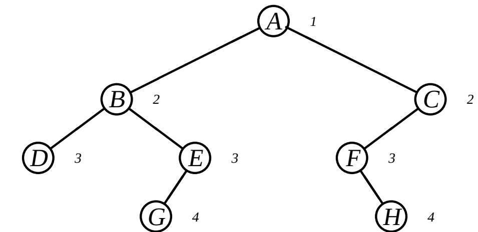
Pré-ordem: A,B,D,E,G,C,F,H

Pós-ordem: D,G,E,B,H,F,C,A

Inordem: D,B,G,E,A,F,H,C

Percurso em largura

(por níveis, da esquerda para a direita)



Percurso: A,B,C,D,E,F,G,H

Implementação recursiva de percursos

```
typedef struct NoArvBin {
  T info;
  struct NoArvBin *esq, *dir;
} NoArvBin, *ArvBin;
```

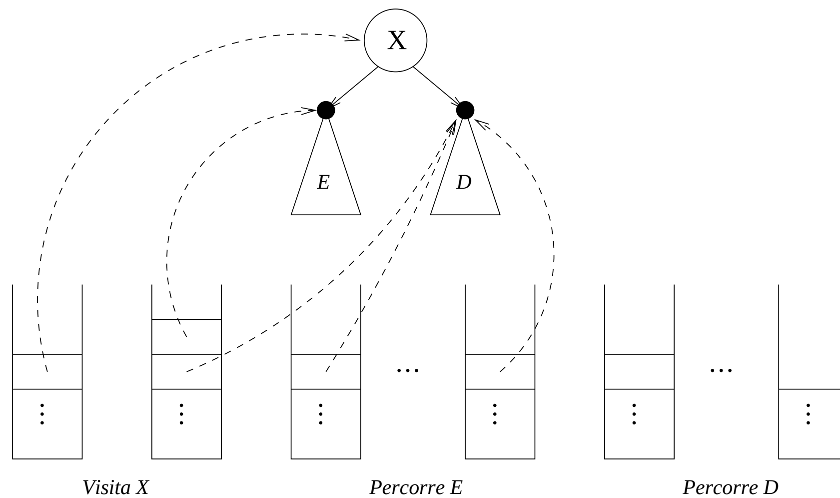
```
void PreOrdem(ArvBin p) {
  if (p!=NULL) {
    Visita(p);
    PreOrdem(p->esq);
    PreOrdem(p->dir);
  }
} /* PreOrdem */
```


Implementação recursiva de percursos (cont.)

```
void InOrdem(ArvBin p) {
  if (p!=NULL) {
    InOrdem(p->esq);
    Visita(p);
    InOrdem(p->dir);
  }
} /* InOrdem */
```

```
void PosOrdem(ArvBin p) {
  if (p!=NULL) {
    PosOrdem(p->esq);
    PosOrdem(p->dir);
    Visita(p);
  }
} /* PosOrdem */
```

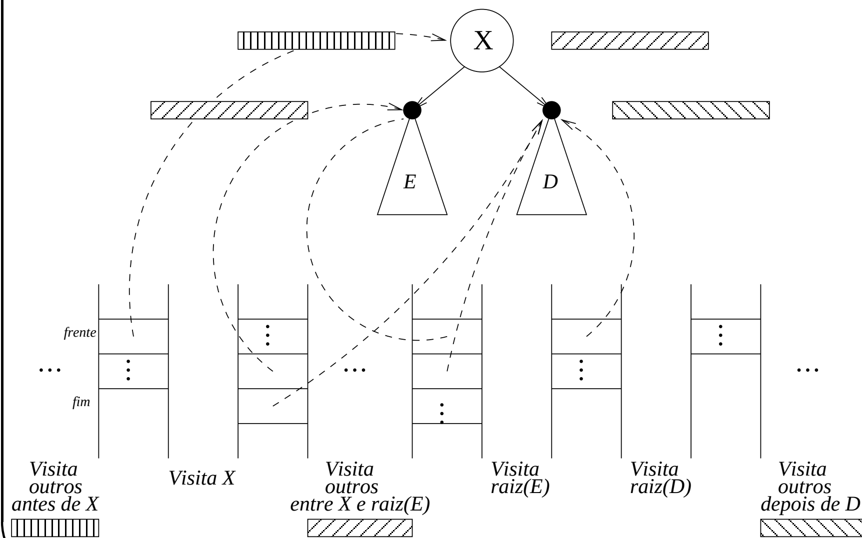
Percurso em pré-ordem, usando uma pilha



Percurso em pré-ordem, usando uma pilha (cont.)

```
void PreOrdem(ArvBin p) {
 Pilha pl;
 InicializaPilha(&pl);
 Empilha(&pl,p);
 do {
  Desempilha(&pl,&p);
  if (p!=NULL) {
   Visita(p);
   Empilha(&pl,p−>dir);
   Empilha(&pl,p−>esq);
  }
 } while (!PilhaVazia(pl));
} /* PreOrdem */
```


Percurso em largura, usando uma fila



Percurso em largura, usando uma fila (cont.)

```
void Largura(ArvBin p) {
 Fila fl;
 InicializaFila(&fl);
 InsereFila(&fl,p);
 do {
  RemoveFila(&fl,&p);
  if (p!=NULL) {
   Visita(p);
   InsereFila(&fl,p−>esq);
   InsereFila(&fl,p−>dir);
  }
 } while (!FilaVazia(fl));
} /* Largura */
```

Comparação dos percursos em pré-ordem e em largura

```
void PreOrdem(ArvBin p) {
  Pilha pl;
  InicializaPilha(&pl);
  Empilha(&pl,p);
  do {
    Desempilha(&pl,&p);
    if (p!=NULL) {
      Visita(p);
      Empilha(&pl,p->dir);
      Empilha(&pl,p->esq);
    }
  } while (!PilhaVazia(pl));
} /* PreOrdem */
```

```
void Largura(ArvBin p) {
  Fila fl;
  InicializaFila(&fl);
  InsereFila(&fl,p);
  do {
    RemoveFila(&fl,&p);
    if (p!=NULL) {
      Visita(p);
      InsereFila(&fl,p->esq);
      InsereFila(&fl,p->dir);
    }
  } while (!FilaVazia(fl));
} /* Largura */
```

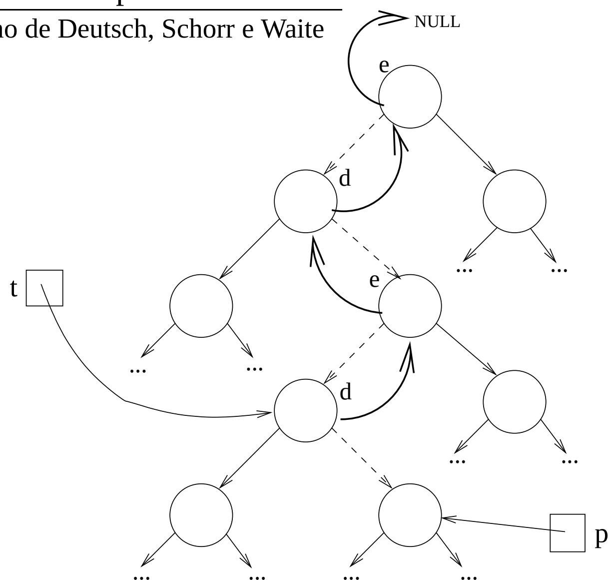
Quase idênticos, exceto a troca de *esquerda* pela *direita*!

Preordem com pilha otimizada

```
void PreOrdem(ArvBin p) {
  Pilha pl;
  Boolean fim = false;
  InicializaPilha(&pl);
  do {
    if (p!=NULL) {
      Visita(p);
      if (p->dir!=NULL)
        Empilha(pl,p->dir);
      p = p->esq;
    } else if (PilhaVazia(pl))
        fim = true
      else
        Desempilha(pl,&p);
  } while (!fim);
} /* PreOrdem */
```


Pré-ordem com pilha "embutida"

Algoritmo de Deutsch, Schorr e Waite




```
void DSW(ArvBin p) {  ArvBin t = NULL;  ArvBin q;  Boolean sobe;
  do {
    while (p!=NULL) {    /* à esquerda */
      PreVisita(p); p->marca = MarcaEsq; q = p->esq; p->esq = t; t = p; p = q;
    }
    sobe = true;
    while (sobe && (t!=NULL)) {
      switch (t->marca) {
        case MarcaEsq:    /* à direita */
          InVisita(t); sobe = false; t->marca = MarcaDir;
          q = p; p = t->dir; t->dir = t->esq; t->esq = q;  break;
        case MarcaDir:    /* sobe */
          PosVisita(t); q = t->dir; t->dir = p; p = t; t = q;  break;
      }
    } /* while */
  } while (t!=NULL);
} /* DSW */
```

Desafios:

- melhorar a pré-ordem com pilha otimizada
- inordem com pilha otimizada
- pós-ordem com pilha otimizada

Representações externas de árvores binárias

- percursos canônicos: inordem e pré (ou pós)-ordem

```
DBGEAFHC
ABDEGCFH
```

- percurso canônico com indicadores de subárvores (pré-ordem):

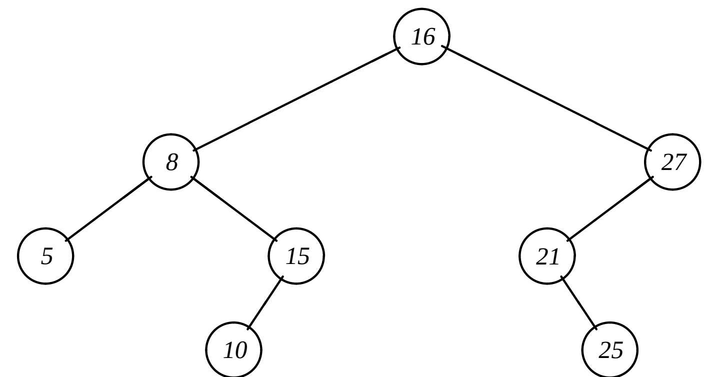
$$A_{11}B_{11}D_{00}E_{10}G_{00}C_{10}F_{01}H_{00}$$

- descrição parentética (inordem):

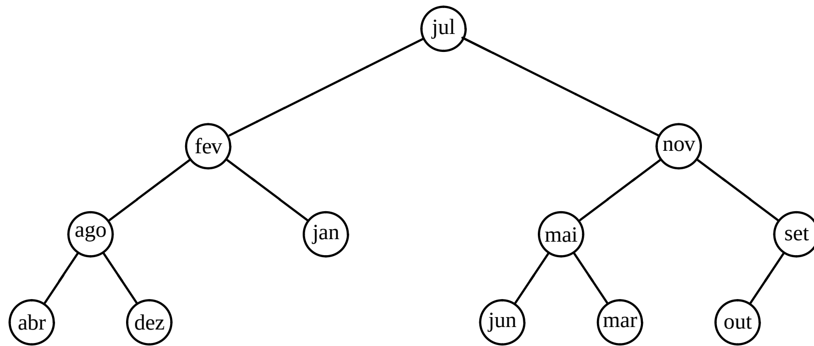
$$((((()D())B((()G())E()))A((()F(()H()))C()))$$

Árvores binárias de busca

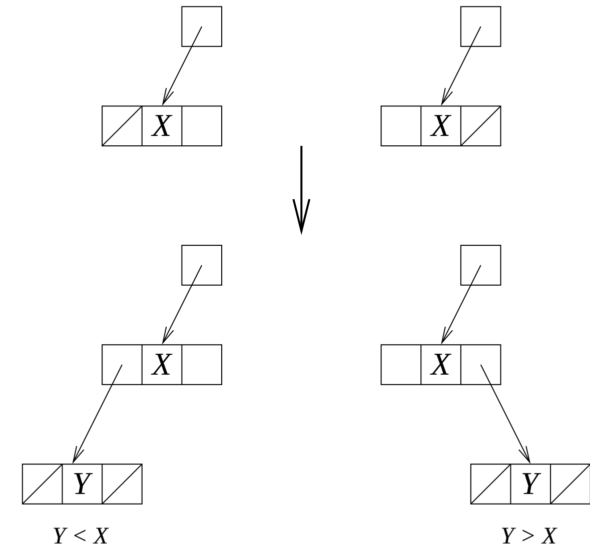
Exemplo de árvores de busca: números



Exemplo de árvores de busca: nomes

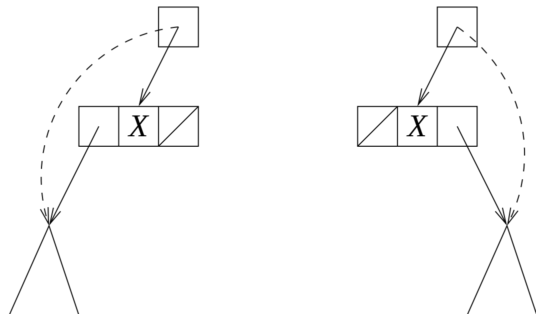


Inserção



Remoção

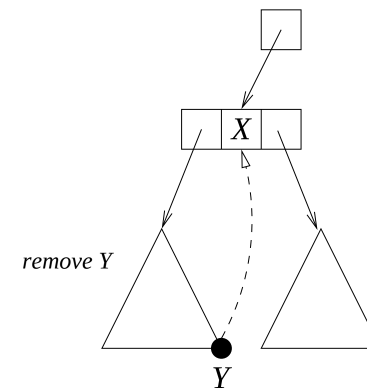
Caso 1: uma das subárvores é vazia



Libera o nó que continha X.

Remoção

Caso 2: as duas subárvores não são vazias



- substitui X por Y – elemento máximo da subárvore esquerda (ou mínimo da direita)
- remove Y da subárvore

Inserção simples numa árvore de busca (versão recursiva)

```
Boolean BuscaInsere(ArvBin *p, T x) {
 /* Devolve 'true' se inseriu; 'false' se já pertencia */
 if ((*p)==NULL) {
  *p = malloc(sizeof(NoArvBin));
  (*p)−>esq = (*p)−>dir = NULL;
  (*p)−>info = x;
  return true;
 }
 if (x==((*p)−>info))
  return false;
 if (x<((*p)−>info))
  return BuscaInsere(&((*p)−>esq),x);
 else
  return BuscaInsere(&((*p)−>dir),x);
} /* BuscaInsere */
```


Inserção simples numa árvore de busca (versão iterativa)

```
Boolean BuscaInsere(ArvBin *p, T x) {
 /* Devolve 'true' se inseriu; 'false' se já pertencia */
 int inf;
 while ((*p)!=NULL) { /* procura o eventual ponto de inserção */
  inf = (*p)−>info;
  if (x==inf) return false;
  if (x<inf)
   p = &((*p)−>esq);
  else
   p = &((*p)−>dir);
 }
 *p = malloc(sizeof(NoArvBin));
 (*p)−>esq = (*p)−>dir = NULL;
 (*p)−>info = x;
 return true;
} /* BuscaInsere */
```


Inserções e remoções em árvores binárias de busca

Problema: A altura da árvore pode crescer muito já que numa árvore com n nós:

- altura máxima n
- altura mínima $\lceil \log_2(n+1) \rceil$

Se $n \approx 1.000$:

- altura máxima 1.000
- altura mínima 10

Se $n \approx 1.000.000$:

- altura máxima $1.000.000$
- altura mínima 20

Pior caso ocorre quando a inserção é feita em ordem (crescente ou descrescente)

Balanceamento de árvores

- algoritmo óbvio não garante balanceamento
- balanceamento perfeito (altura mínima):
  - busca: $\mathrm{O}(\log n)$
  - inserção: $\mathrm{O}(n)$ (!)
- balanceamento aproximado:
  - árvores AVL – busca, inserção e remoção: $\mathrm{O}(\log n)$
  - árvores rubro-negras – busca, inserção e remoção: $\mathrm{O}(\log n)$

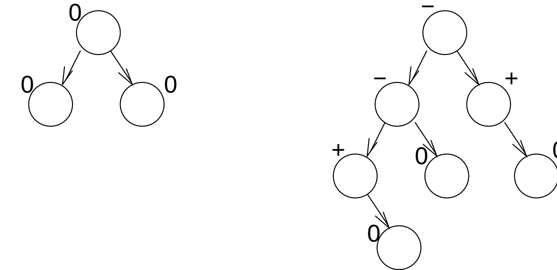
Árvores de altura balanceada (AVL)

(G. M. Adel'son-Vel'skiĭ e E. M. Landis)

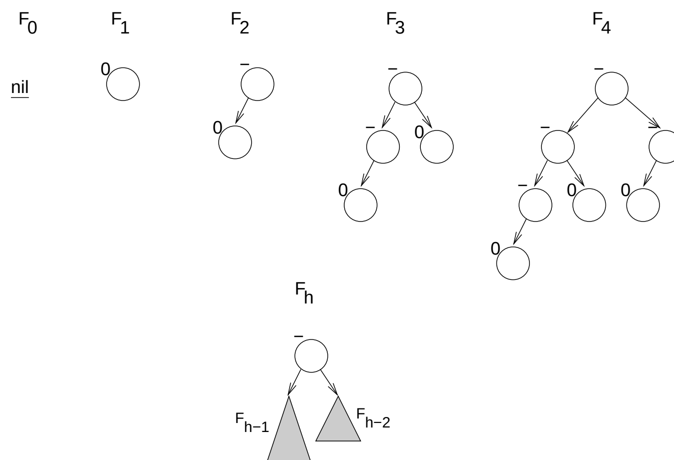
Uma árvore binária de busca é do tipo AVL se para todos os seus nós a diferença de alturas entre as subárvores esquerda e direita é no máximo 1, em valor absoluto.

A diferença entre as alturas direita e esquerda é chamada *fator de balanceamento*.

Exemplos de árvores AVL



Pior caso de desbalanceamento: árvores de Fibonacci



Propriedades

- Número de nós de F_h:

$$N(h) = N(h-1) + N(h-2) + 1, \quad h \geq 2$$

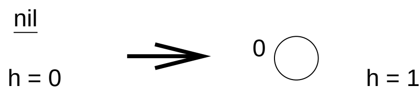
$$N(h) = f_{h+2} - 1$$

- Altura aproximada de F_h ($f_i \approx ((1+\sqrt(5))/2)^i/\sqrt(5)$):

$$1,44 \log_2(n+2)$$

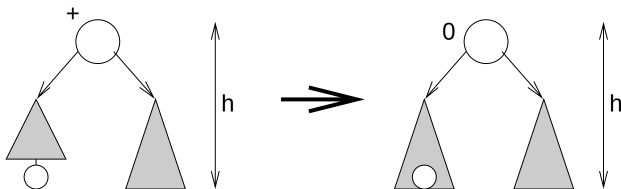
Inserção em árvores AVL

Caso 1: Inserção em árvore vazia



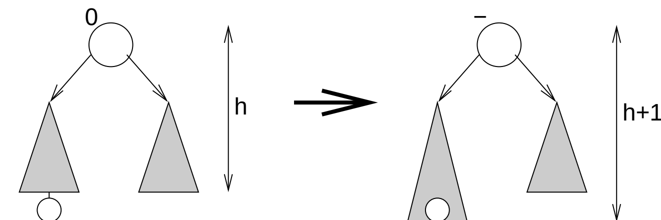
A altura final aumenta.

Caso 2: Inserção do lado mais baixo



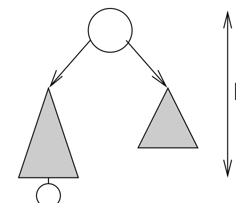
A altura final permanece inalterada (o processo de inserção pára).

Caso 3: Inserção quando as duas alturas são iguais



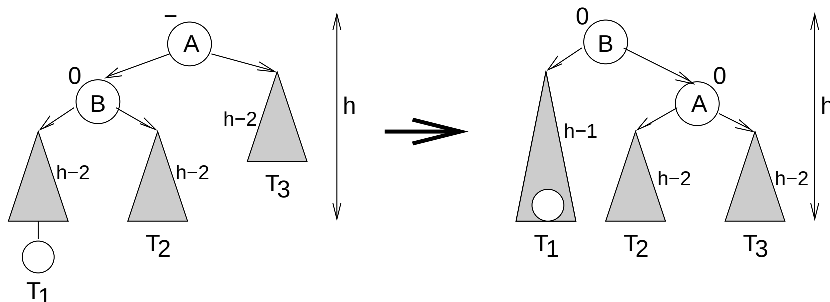
A altura final aumenta.

Caso 4: Inserção do lado mais alto



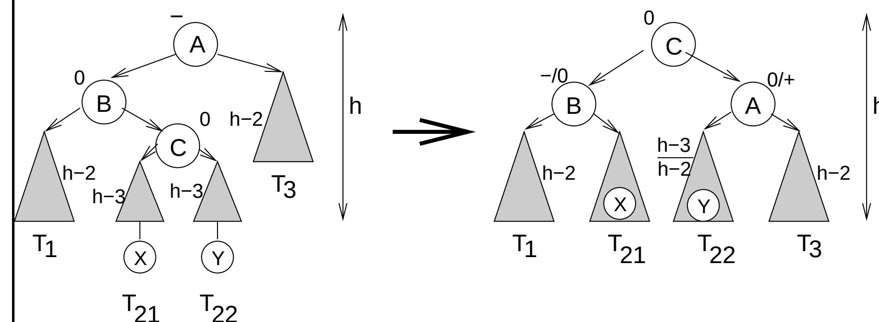
Deixa de ser AVL. Há dois subcasos:

(4a): Caminho LL (esquerdo, esquerdo) – rotação simples



A altura final permanece inalterada (o processo de inserção pára).

(4b): Caminho LR (esquerdo, direito) – rotação dupla



O nó inserido pode ser X ou Y. Quando h=2, as árvores T_1 e T_3 são vazias, e o nó inserido é o próprio nó C; neste caso as árvores T_{21} e T_{22} são vazias.

A altura final permanece inalterada (o processo de inserção pára)

```
Boolean BuscaInsere(ArvAVL *p, T x, Boolean *h) {
 /* Devolve 'true' ou 'false' conforme houve ou não inserção;
    se houve inserção, 'h' indica se houve aumento da altura. */
 if (*p==NULL) {
  *p = malloc(sizeof(NoArvAVL));
  (*p)−>esq = (*p)−>dir = NULL; (*p)−>info = x;
  (*p)−>bal = zero;  *h = true;
  return true;
 } else {
  T info = (*p)−>info;
  if (x==info)
   return false;
  else if (x<info) {  /* desce à esquerda */
   Boolean res = BuscaInsere(&((*p)−>esq),x,h);
   if (!res)
    return false;
```

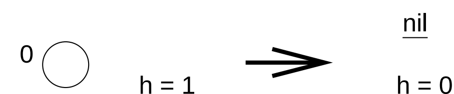


```
   if (*h) {        /* aumento de altura */
    ArvAVL p1, p2;
    switch ((*p)−>bal) {
     case mais: (*p)−>bal = zero; *h = false; break;
     case zero: (*p)−>bal = menos; break;
     case menos:
      p1 = (*p)−>esq;
      if (p1−>bal==menos) { ... }  /* Rotação simples LL */
      else  {     ...    }                  /* Rotação dupla LR */
      p1−>bal = zero;  *h = false;
      break;
    }
   }
   return true;
  } else {     ...     }         /* desce à direita − análogo */
}
```


Remoção em árvores AVL

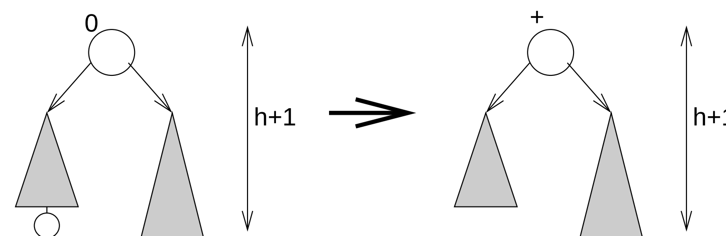
1. Transformação em remoção de uma folha - três casos:
   - o nó tem grau zero: já é uma folha
   - o nó tem grau um: pela propriedade AVL, a sua única subárvore é necessariamente constituída por uma folha, cujo valor é copiado para o nó pai; o nó a ser eliminado é a folha da subárvore
   - o nó tem grau dois: o seu valor é substitído pelo maior valor contido na sua subárvore esquerda (ou o menor valor contido na sua subárvore direita); o nó que continha o menor (ou maior) valor copiado tem necessariamente grau zero ou um, recaindo num dos casos anteriores.
2. Remoção propriamente dita

Caso 1: Remoção de uma folha



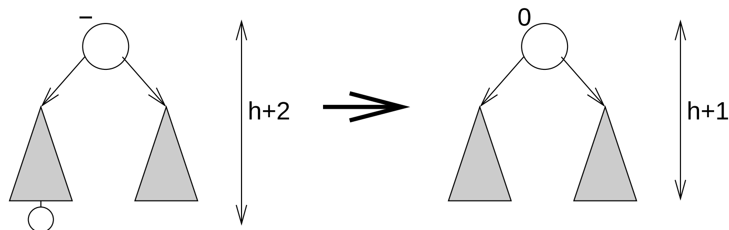
A altura diminui (o processo continua).

Caso 2: Alturas iguais



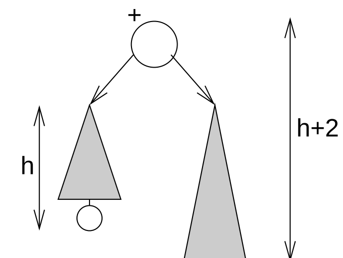
A altura permanece inalterada (o processo pára).

Caso 3: Remoção do lado mais alto



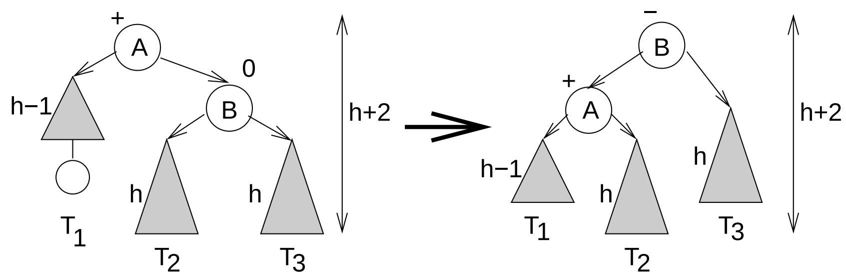
A altura diminui (o processo continua).

Caso 4: Remoção do lado mais baixo



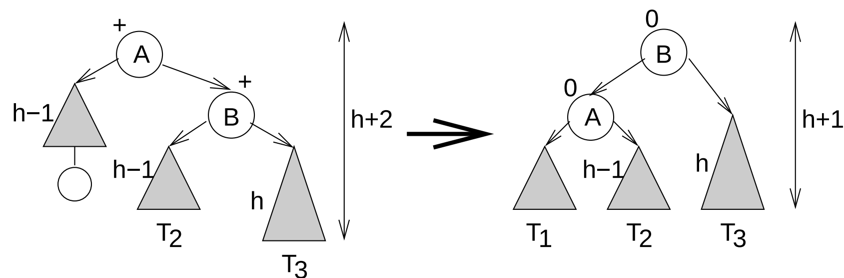
Há três subcasos, dependendo do fator de balanceamento do filho direito da raiz:

(4a): Fator "0" (rotação simples RR)



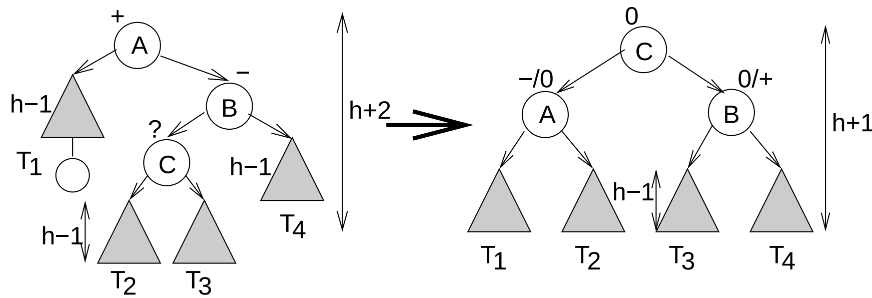
A altura permanece inalterada (o processo pára).

(4b): Fator "+" (rotação simples RR)



A altura diminui (o processo continua).

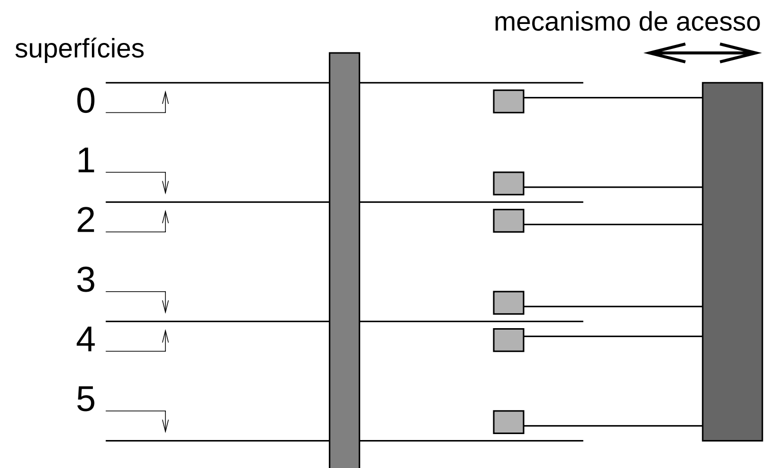
(4c): Fator "-" (rotação dupla RL)



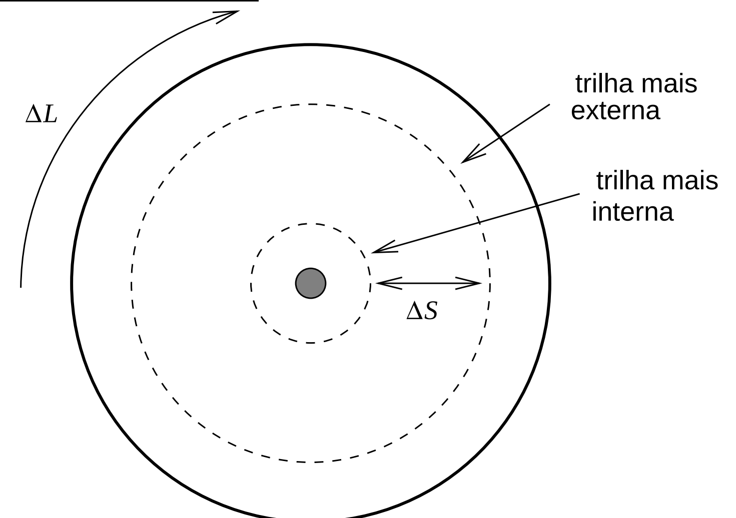
A altura diminui (o processo continua).

Árvores do tipo *B* (*B-trees*)

Discos magnéticos (1)



Discos magnéticos (2)



Problemas com arquivos em disco

- Tempo de acesso:
  - tempo de busca (*seek*): ΔS
  - tempo de latência: ΔL
  - tempo de transferência de dados: ΔT
- Número de acessos: altura da árvore – $log_2 n$ não é mais aceitável
- Solução: $log_k n$, com $k >> 2$

Definição de árvores B

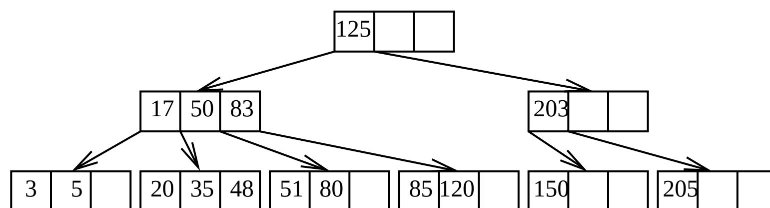
T é uma *árvore B de ordem* $b \geq 2$ se

1. todas as folhas de T têm o mesmo nível;
2. cada nó interno tem um número variável r de registros e $r+1$ de filhos, onde

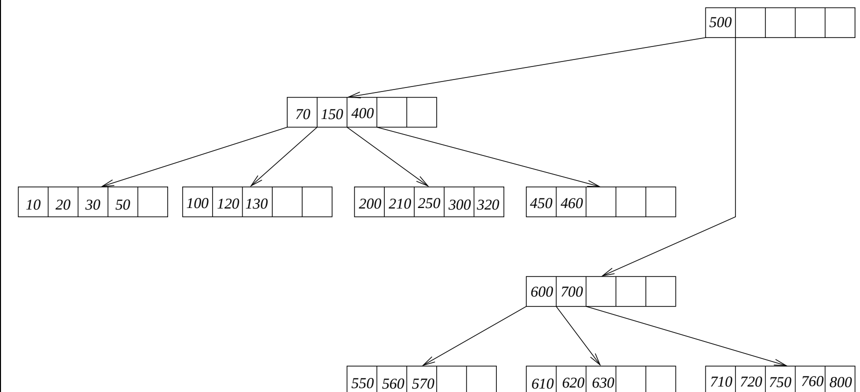
$$\begin{cases} \lfloor b/2 \rfloor \leq r \leq b & \text{se nó} \neq \text{raiz} \\ 1 \leq r \leq b & \text{se nó = raiz} \end{cases}$$

3. cada folha tem um número variável r de registros obedecendo à mesma restrição do item anterior;
4. os campos de informação contidos nos registros obedecem à propriedade de árvores de busca.

Exemplo de árvore B de ordem 3



Exemplo de árvore B de ordem 5



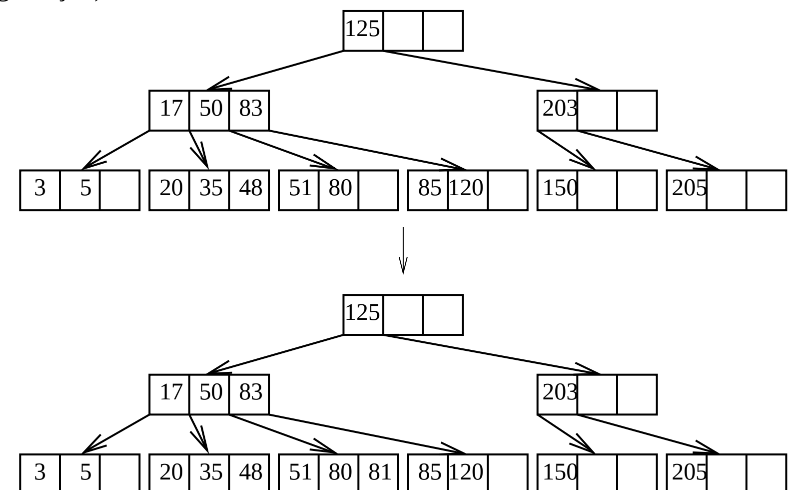
Números mínimos e máximos de registros

(árvore B de ordem 255)

	mínimo		máximo	
nível	nós	registros	nós	registros
1	1	1	1	1×255
2	2	2×127	256	256×255
3	2×128	$2 \times 128 \times 127$	256^2	$256^2 \times 255$
4	2×128^2	$2 \times 128^2 \times 127$	256^3	$256^3 \times 255$
Total	33.027	4.194.303	16.843.009	4.294.967.295

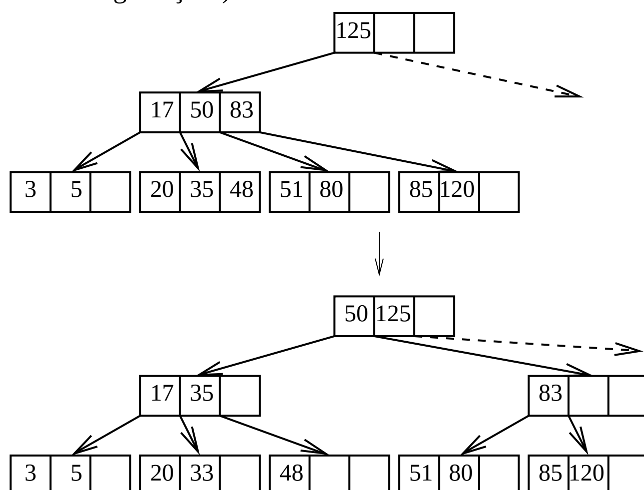
Exemplo de inserção:

81 (folha com espaço: h leituras e uma gravação)



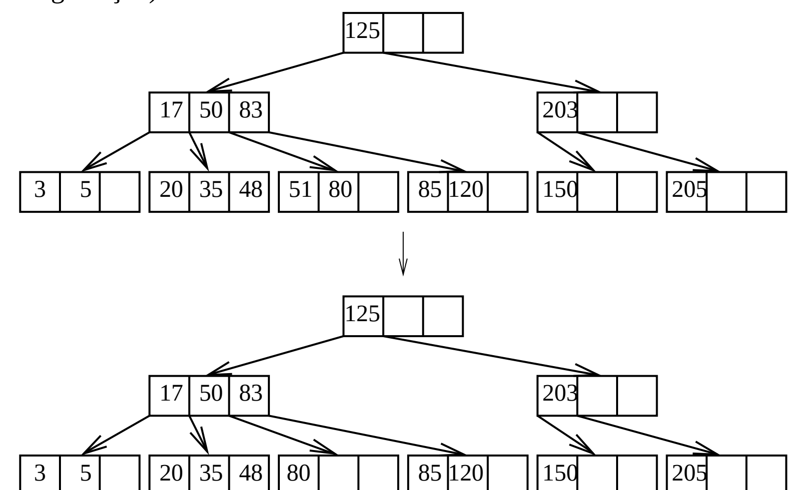
Exemplo de inserção:

33 (estouro da folha: no máximo h leituras e $2h+1$ gravações)

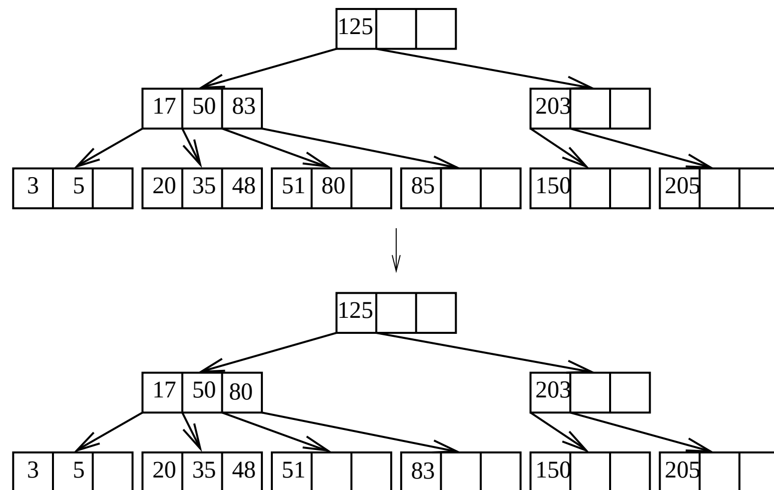


Exemplo de remoção:

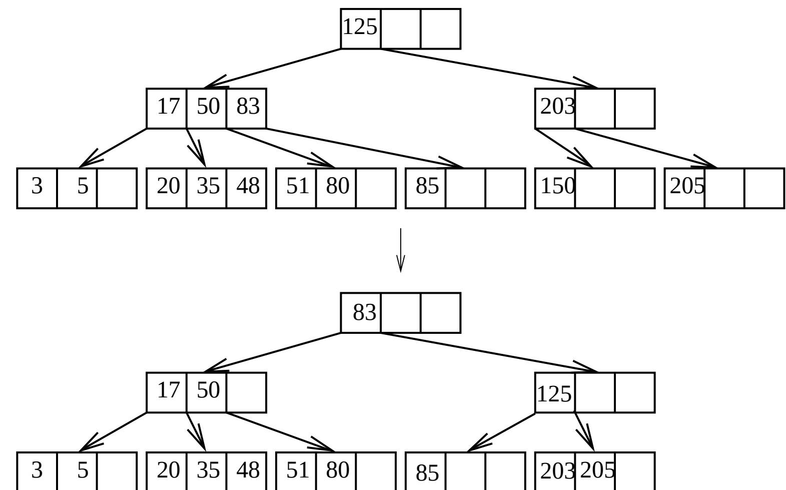
51 (folha acima do mínimo: h leituras e uma gravação)



Exemplo de remoção: 85 (folha abaixo do mínimo, com empréstimo: no máximo $h + 2$ leituras e 3 gravações)



Exemplo de remoção: 150 (folha abaixo do mínimo, sem empréstimo: no máximo $3h - 2$ leituras e $2h - 1$ gravações)



Implementação na memória: tipos

```
#define ORDEM 255

typedef struct NoArvB * ArvB;

typedef struct NoArvB {
  int numregs;
  ArvB filhos[ORDEM+1];
  T info[ORDEM];
} NoArvBin;
```


Busca e inserção (1)

```
Boolean InsereRecArvB(ArvB *p,ArvB *s, T *x, Boolean *h) {
/* Devolve 'false' se o valor de 'x' já ocorre na árvore */
  int i;
  Boolean inseriu;
  if (p==NULL) {
    *h = true; *s = NULL; return true;
  }
  i = IndiceArvB(p,x);
  if ((i<((*p)->numregs)) and (x==((*p)->info)[i])) {
    *h = false; return false;
  }
  ...
```

Busca e inserção (2)

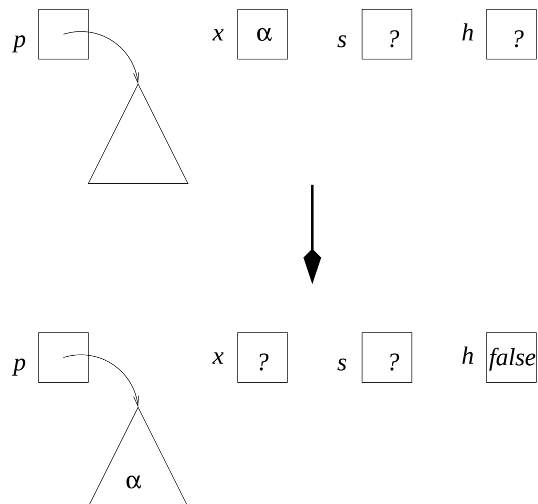
```
...
inseriu = InsereRecArvB(&((*p)−>filhos[i]),s,x,h);
if (*h) {
  [InsereInfoArvB(p,s,x,i)];
  ((*p)−>numregs)++;
  if (((*p)−>numregs<=ORDEM))
    *h = false;
  else {
    [QuebraNoArvB(p,s,x)]; h = true;
  }
}
return inseriu;
} /* InsereRecArvB */
```


Busca e inserção (função inicial)

```
Boolean InsereArvB(ArvB *p, T *x) {
/* Devolve 'false' se o valor de 'x' já ocorre na árvore 'p' */
  Boolean h;
  ArvB q,s;
  Boolean inseriu = InsereRecArvB(p,&s,x,&h);
  if (h) {
    q = (ArvB)malloc(sizeof(NoArvB));
    q−>numregs = 1;
    (q−>filhos)[0] = *p;  (q−>filhos)[1] = s;
    (q−>info)[0] = *x;
  *p = q;
  }
  return inseriu;
} /* InsereArvB */
```


Explicação geral

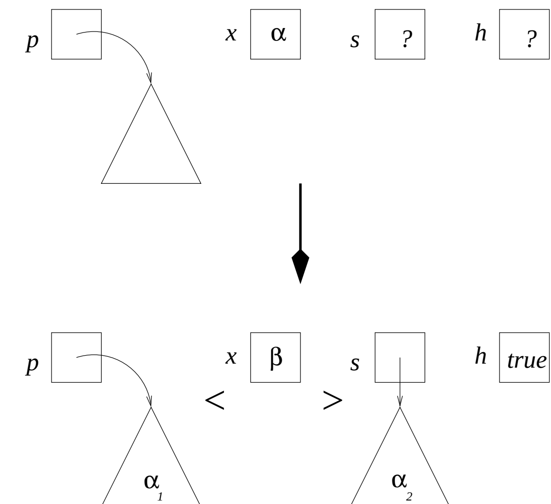
Caso 1: valor devolvido h = *false*



Não há mais nada para modificar.

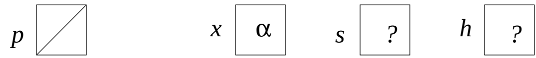
Explicação geral

Caso 2: valor devolvido h = *true*

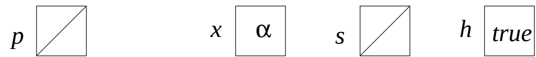


A modificação é propagada para cima (α está numa das subárvores).

Inserção em subárvore vazia

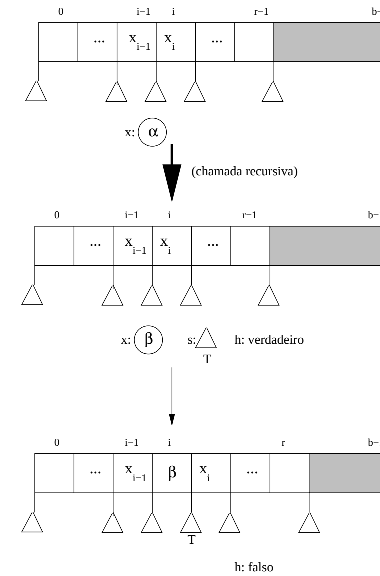




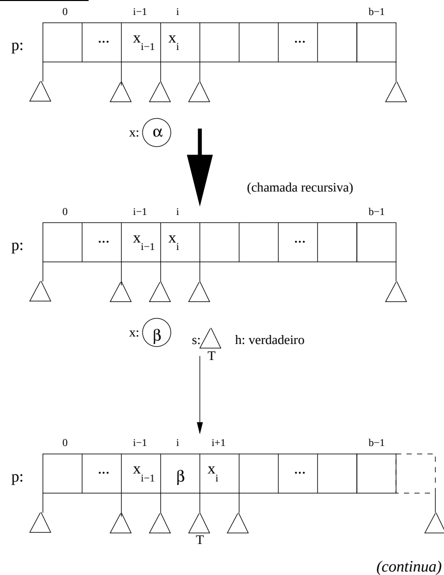


Propaga a inserção.

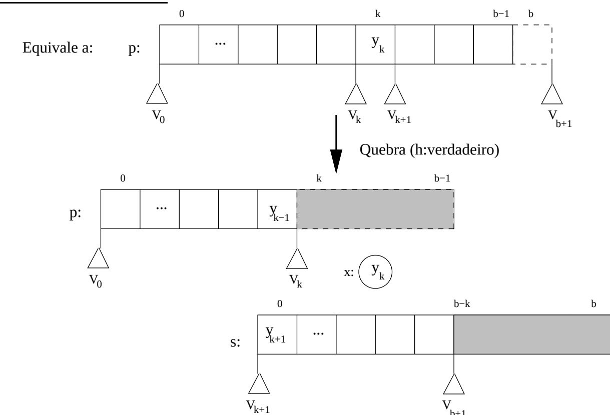
Inserção com $r < b$: Não haverá propagação



Inserção com $r = b$: Haverá quebra e propagação



Inserção com $r = b$ (cont.)

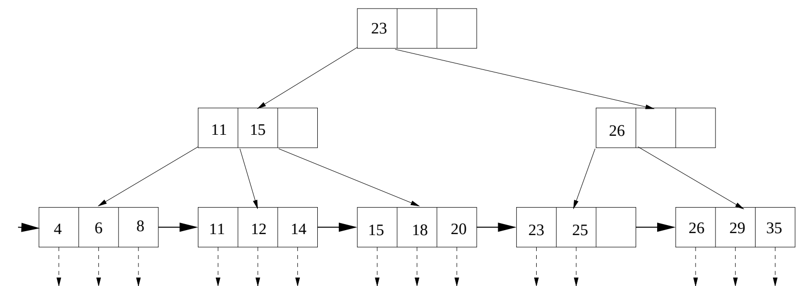


$k = \lceil b/2 \rceil + 1$

Variantes de árvores B

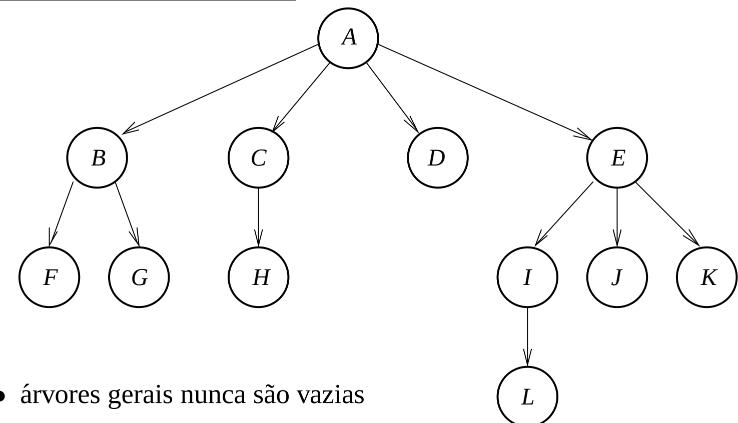
- árvores $B^{\star}$: $2/3$ dos campos de informação preenchidos (exceto a raiz)
- árvores B^{+}:
  - nós internos com chaves
  - (chaves,valores) nas folhas
  - regra de descida
    - subárvore esquerda: menor
    - subárvore direita: maior ou igual
  - apontadores em lugar de valores
  - percurso fácil em ordem

Exemplo de árvore B^{+}



Árvores gerais

Exemplo de árvore geral



- árvores gerais nunca são vazias
- as subárvores são ordenadas: primeira, segunda, etc
- o número de subárvores pode ser qualquer, inclusive zero
- conceitos naturais: grau, filhos, pai, descendente, altura, etc

Representação de árvores gerais

```
#define GRAU_MAX 10
typedef struct
      NoArvGeral *ArvGeral;
typedef struct NoArvGeral {
  T info;
  int grau;
  ArvGeral filhos[GRAU_MAX];
} NoArvGeral;

...

p = malloc(sizeof(NoArvGeral));

...
```

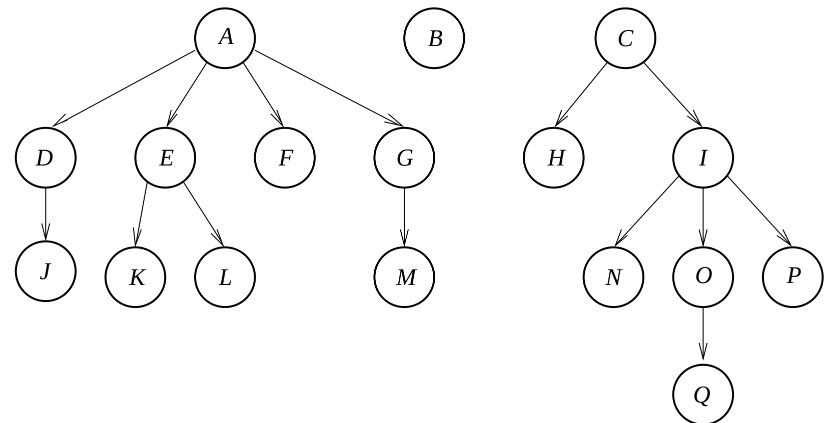
```
typedef struct
      NoArvGeral *ArvGeral;
typedef struct NoArvGeral {
  T info;
  int grau;
  ArvGeral filhos[1];
} NoArvGeral;

...

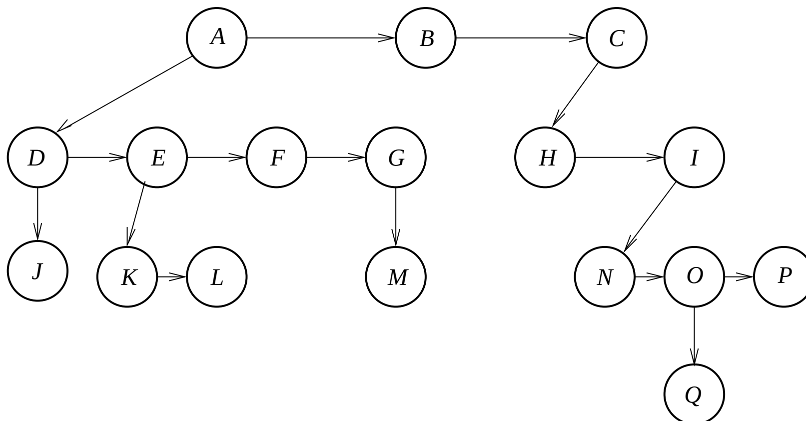
p = malloc(sizeof(NoArvGeral)+
          (grau−1)*sizeof(ArvGeral));
...
```


Floresta: seqüência de árvores gerais

Exemplo:



Floresta representada como árvore binária



Definição da transformação

A árvore binária $\mathbf{B}(F)$ que representa uma floresta $F = (T_1, T_2, \ldots, T_m)$ é definida por:

(a) árvore binária vazia se F é uma floresta vazia ($m = 0$);

(b) árvore binária cuja raiz contém a mesma informação da raiz de T_1; cuja subárvore esquerda é dada por $\mathbf{B}((T_{11}, T_{12}, \ldots, T_{1m_1}))$ onde $(T_{11}, T_{12}, \ldots, T_{1m_1})$ é a floresta das subárvores de T_1; e cuja subárvore direita é dada por $\mathbf{B}((T_2, \ldots, T_m))$.

Percursos em profundidade de florestas: $F = (T_1, T_2, \ldots, T_m)$

- *Pré-ordem de florestas:*

  Visitar a raiz de T_1
  Percorrer a floresta F_1 em pré-ordem de florestas
  Percorrer a floresta $(T_2, \ldots, T_m)$ em pré-ordem de florestas

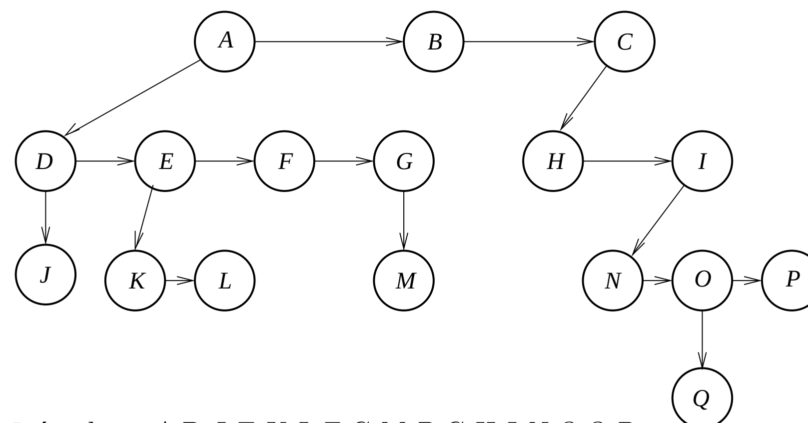
- *Pós-ordem de florestas:*

  Percorrer a floresta F_1 em pós-ordem de florestas
  Percorrer a floresta $(T_2, \ldots, T_m)$ em pós-ordem de florestas
  Visitar a raiz de T_1

- *Inordem de florestas:*

  Percorrer a floresta F_1 em inordem de florestas
  Visitar a raiz de T_1
  Percorrer a floresta $(T_2, \ldots, T_m)$ em inordem de florestas

Exemplo de percursos de florestas



Pré-ordem: A,D,J,E,K,L,F,G,M,B,C,H,I,N,O,Q,P

Pós-ordem: J,L,K,M,G,F,E,D,Q,P,O,N,I,H,C,B,A

Inordem: J,D,K,L,E,F,M,G,A,B,H,N,Q,O,P,I,C

Propriedades de percursos de florestas

- percurso de uma floresta F produz o mesmo resultado que o percurso (binário) da árvore **B**(F).
- pré-ordem de florestas é semelhante à pré-ordem de árvores binárias
- inordem de florestas é semelhante à pós-ordem de árvores binárias
- pós-ordem de florestas não tem uma interpretação natural

Desafio:

Elabore um algoritmo para percurso em largura de árvores gerais sob representação binária.

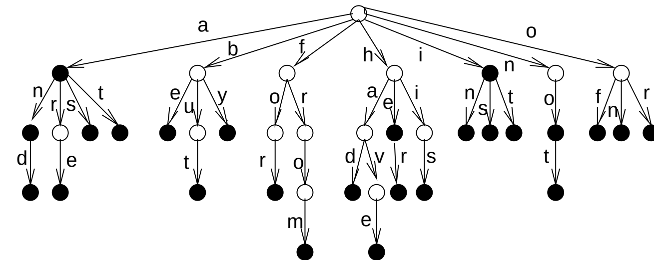
Árvores digitais

Representação de conjuntos de cadeias de caracteres

Exemplo:

a	at	from	his	no
an	be	had	i	not
and	but	have	in	of
are	by	he	is	on
as	for	her	it	or

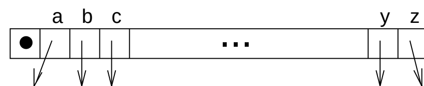
Árvore digital



Nós da forma ● indicam o fim de uma cadeia.

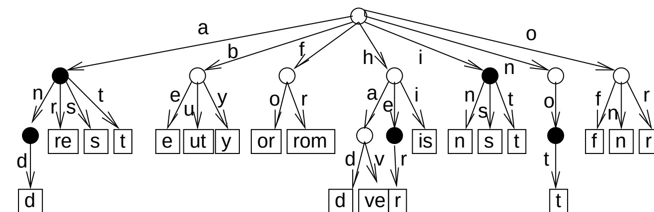
39 nós, 20 folhas, 19 nós não folhas, 25 nós com ●

Implementação de árvores digitais



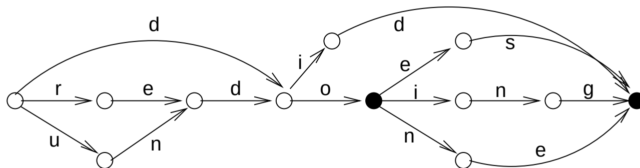
- desperdício de memória: dos $39 \times 26 = 1014$ campos, 38 são não nulos
- simplificando as folhas: dos $19 \times 26 = 494$ campos, 38 são não nulos
- uso de subcadeias (transparência seguinte): dos $12 \times 26 = 312$ campos, 31 não nulos

Árvore digital modificada



Autômato finito acíclico

Exemplo: 15 formas dos verbos ingleses: `do`, `redo` e `undo`



- 11 nós internos: $11 \times 26 = 286$ campos, 16 não nulos
- se fosse árvore: $26 \times 26 = 676$ campos, 37 não nulos
- construção muito mais complicada

Filas de prioridade

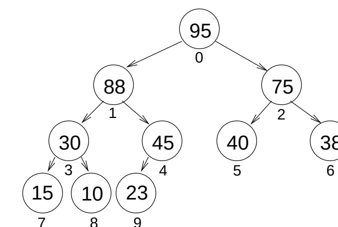
Fila de prioridade

Árvore binária com as propriedades:

- a árvore é *completa* ou *quase completa*;
- em cada nó da árvore, o valor da chave é maior do que os valores das chaves dos filhos.

Exemplo e implementação

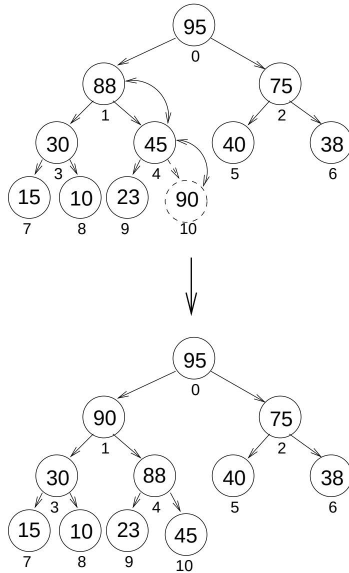
Exemplo:



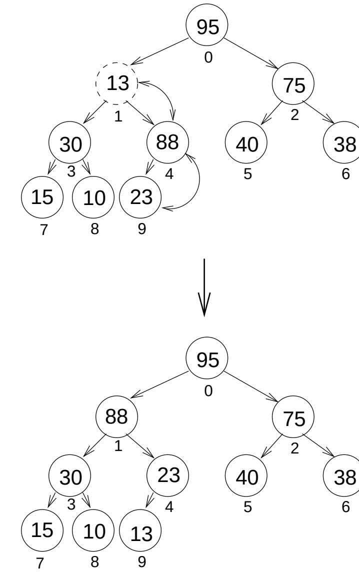
Implementação (*heap*):

95	88	75	30	45	40	38	15	10	23
0	1	2	3	4	5	6	7	8	9

Operação de *subida*: $O(\log_2 N)$



Operação de *descida*: $O(\log_2 N)$



Operação *sobe* em C

```
#define TAM_MAX 50

typedef struct {
  T vetor[TAM_MAX];
  int tam;
} Heap;
```

```
void Sobe(Heap *h, int m) {
  int j = (m−1)/2;
  T x = (*h).vetor[m];

  while ((m>0) && ((*h).vetor[j]<x)) {
    (*h).vetor[m] = (*h).vetor[j];
    m = j;
    j = (j−1)/2;
  }
  (*h).vetor[m] = x;
} /* Sobe */
```


Operação *desce* em C

```
void Desce(Heap *h, int m) {
  int k = 2*m+1;
  T x = (*h).vetor[m];
  while (k<(*h).tam) {
    if ((k<((*h).tam)−1) && ((*h).vetor[k]<(*h).vetor[k+1]))
      k++;
    if (x<(*h).vetor[k]) {
      (*h).vetor[m] = (*h).vetor[k];
      m = k;
      k = 2*k+1;
    } else
      break;
  }
  (*h).vetor[m] = x;
} /* Desce */
```

Construção de *heaps*: $O(N \log_2 N)$

```
void ConstroiHeap1(Heap *h) {
  int i;
  for (i=1; i<(*h).tam; i++)
    Sobe(h,i);
} /* ConstroiHeap1 */
```

```
void ConstroiHeap2(Heap *h) {
  int i;
  for (i=((*h).tam-2)/2; i>=0; i--)
    Desce(h,i);
} /* ConstroiHeap2 */
```


Inserção e remoção em *heaps*: $O(\log_2 N)$

```
void InsereHeap(Heap *h, T x) {
  vetor[(*h).tam] = x;
  ((*h).tam)++;
  Sobe(h,((*h).tam)-1);
} /* InsereHeap */
```

```
void RemoveHeap(Heap *h, T *x) {
  *x = (*h).vetor[0];
  ((*h).tam)--;
  (*h).vetor[0] =
        (*h).vetor[(*h).tam];
  Desce(h,0);
} /* RemoveHeap */
```


Algoritmo de ordenação *heapsort*

```
void HeapSort(Heap *h) {
  int i, n = (*h).tam;
  for (i=(n-2)/2; i>=0; i--) /* constrói heap */
    Desce(h,i);
  for (i=n-1; i>0; i--) {   /* ordena */
    T t = (*h).vetor[0];
    (*h).vetor[0] = (*h).vetor[i];
    (*h).vetor[i] = t;
    (*h).tam--;
    Desce(h,0);
  }
  (*h).tam = n;
} /* HeapSort */
```

Número de operações: $O(N \log_2 N)$ (um dos algoritmos ótimos)

Espalhamento

(*hashing* ou *scattering*)

Tabelas de espalhamento

		1	2	3
	0			
	1			
	2			
$f($'joão'$) \rightarrow$	3	joão		
	4			
	5			
	6			

$b \geq 1$ linhas e $s \geq 1$ colunas ($b = 7$, $s = 3$)

Exemplo de tabelas de espalhamento: $b = 26$ e $s = 2$

	1	2
0	antônio	átila
1		
2	carlos	célio
3	douglas	
4	ernesto	estêvão
5		
...		
24		
25	zoroastro	

Função (muito ingênua!) de espalhamento: índice da primeira letra ($a : 0$, $b : 1$, ...).

Virtudes e poblemas

- Virtudes
  - simplicidade
  - busca muito rápida (se a função de espalhamento for eficiente)
- Problemas
  - escolha da função de espalhamento
  - tratamento de colisões
  - estouro da tabela

Técnicas para construção de funções de espalhamento

Propriedades desejáveis:

- eficiência de cálculo
- bom espalhamento

Técnicas:

- espalhamento mínimo perfeito
- espalhamento pseudo-aleatório

Divisão

O nome, tratado como um número na base 26, é dividido por um número p relativamente primo $f(x) = x \textbf{ mod } p$. Exemplo, para $p = 51$ teríamos:

$$\begin{aligned} f(\texttt{carlos}) &= (\,((((2 \times 26 + 0) \times 26 + 17) \times 26 + 11) \\ &\quad \times 26 + 14) \times 26 + 18\,) \textbf{ mod } 51 \\ &= 24.069.362 \textbf{ mod } 51 = 14 \end{aligned}$$

Seleção de algarismos e meio-do-quadrado

O nome é tratado como uma seqüência de algarismos ou de *bytes* ou de *bits*, e uma subseqüência é selecionada para representar o índice. Por exemplo, suponhamos que todos os nomes são representados como a seqüência de dígitos $x = d_0 d_1 \cdots d_{11}$ em alguma base conveniente; uma escolha seria $f(x) = d_3 d_5 d_9$.

Exemplo: '`carlos`'. A sua representação poderia ser 020017111418. Supusemos que cada letra é indicada por dois dígitos que indicam a posição no alfabeto, ou seja 00 para '`a`', 01 para '`b`', etc. Teríamos então $f(\texttt{carlos}) = 074$.

Freqüentemente, antes de fazer a seleção, é calculado o quadrado do identificado (tratado como número); é o método "meio-do-quadrado" (*mid-square*).

Dobramento (*folding*)

o nome é tratado como uma seqüência de algarismos ou de *bytes* ou de *bits*, e algumas subseqüências são combinadas por operações convenientes para produzir um índice. Por exemplo, suponhamos que todos os nomes são representados como a seqüência de dígitos $x = d_0 d_1 \cdots d_{11}$ em alguma base conveniente; uma escolha seria $f(x) = d_3 d_5 d_9 \oplus d_8 d_7 d_{10}$, onde $\oplus$ denota a operação de *ou exclusivo bit a bit*.

Colisões: endereçamento aberto

Busca sistemática de outras entradas disponíveis na tabela:

- reespalhamento linear
- reespalhamento quadrático
- reespalhamento duplo

Exemplos usam: `antônio`, `carlos`, `douglas`, `célio`, `armando`, `zoroastro`, `átila`, `alfredo` (nesta ordem).

Reespalhamento linear

Procura posição livre com $(f(x)+i) \mathbf{mod}\, b, (i=0,1,\cdots)$.

0	antônio
1	armando
2	carlos
3	douglas
4	célio
5	átila
6	alfredo
7	
...	
25	zoroastro

Reespalhamento quadrático

$(f(x)+i^2) \mathbf{mod}\, b, (i=0,1,\cdots)$

0	antônio
1	armando
2	carlos
3	douglas
4	átila
5	
6	célio
7	
8	
9	alfredo
...	
25	zoroastro

Reespalhamento duplo

$(f(x)+i \times g(x)) \mathbf{mod}\, b, (i=0,1,\cdots)$

Exemplo:
$g(x) = (c \,\mathbf{mod}\, 3) + 1$

(c *segunda letra)*

0	antônio
1	
2	carlos
3	douglas
4	célio
5	
6	armando
7	
8	átila
9	alfredo
...	
25	zoroastro

Remoção: lápides (*tombstones*)

0	antônio
1	+++++++
2	carlos
3	douglas
4	átila
5	
6	célio
7	
8	
9	alfredo
...	
25	zoroastro

Eficiência com endereçamento aberto

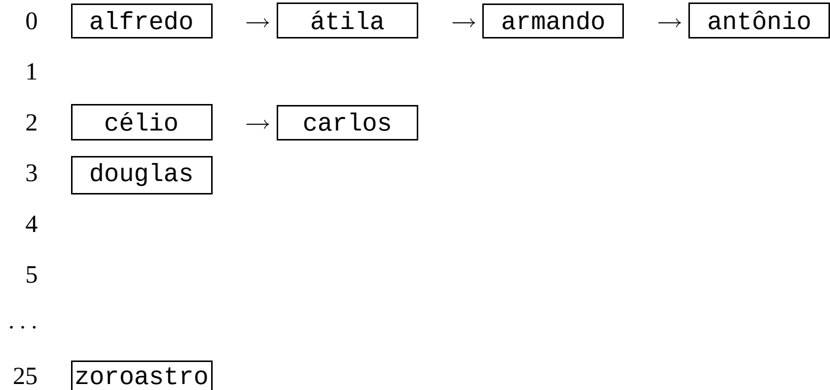
Número médio de comparações para encontrar um elemento: $(2-\alpha)/(2-2\alpha)$
onde
$\alpha = n/b$ (*fator de carga*, $0 \leq \alpha \leq 1$)
n é o número de entradas
b é o tamanho da tabela

Exemplo: tabela com 1000 entradas.

100	1,06
200	1,13
300	1,21
400	1,33
500	1,50
600	1,75
700	2,17
800	3,00
900	5,50
950	10,5

Colisões: encadeamento (*chaining*)

Uso de listas ligadas:



As listas poderiam ser oredenadas.

Eficiência com encadeamento

Número médio de comparações para encontrar um elemento: $1+\alpha/2$
onde
$\alpha = n/b$ (*fator de carga*, $\alpha > 0$)
n é o número de entradas
b é o tamanho da tabela

Exemplo: tabela com 1000 entradas.

100	1,05
200	1,10
400	1,20
500	1,25
1000	1,50
2000	2,00

Listas generalizadas

Conceito e exemplos

Forma da lista generalizada:

$$(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

onde α_i denota um átomo ou uma lista generalizada (definição recursiva).

Exemplos:

A: ((4,7),(4,7,(8)))
B: ((1,4),(7,8))
C: (3,B,B)
D: (5,8,D)
E: ()

As listas *A*, *B*, *C*, *D* e *E* têm, respectivamente, 2, 2, 3, 3 e 0 elementos.

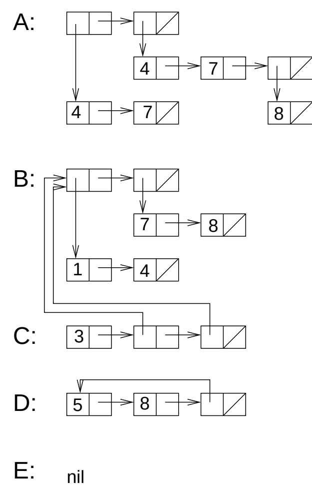
Substituição (ou expansão)

As listas *C* e *D* podem ser expandidas com as definições correspondentes:

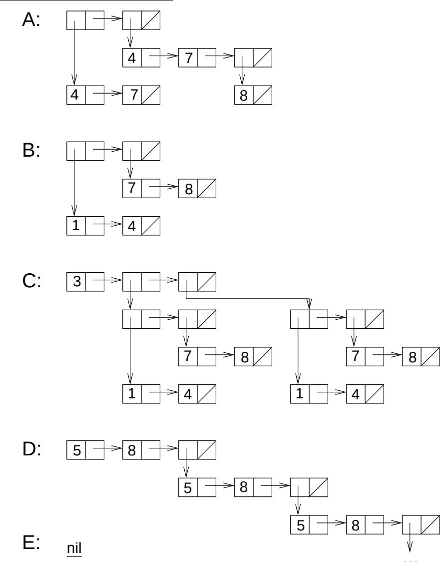
C: (3,((1,4),(7,8)),((1,4),(7,8)))
D: (5,8,(5,8,(5,8,(...))))

Obs.: A lista D, tem três elementos, mas inclui um número infinito de inteiros, por ser recursiva.

Implementação compartilhada



Implementação com cópia



Representação em C

```
typedef struct RegListaGen *ListaGen;

typedef struct RegListaGen {
  ListaGen prox;
  Boolean eAtomo;
  union {
    int atomo;       /* caso de 'eAtomo' verdadeiro */
    ListaGen lista;  /* caso de 'eAtomo' falso */
  } info;
} RegListaGen;
```


Exemplo de manipulação: contagem de átomos.

```
int ContaAtomos(ListaGen p) {
  int s = 0;
  while (p!=NULL) {
    if (p->eAtomo)
      s++;
    else
      s += ContaAtomos(p->info.lista);
    p = p->prox;
  }
  return s;
} /* ContaAtomos */
```

Problemas com compartilhamento: contagem repetida (pode ser intencional) ou repetição infinita.

Representação alternativa em C

```
typedef struct RegListaGen *ListaGen;

typedef struct RegListaGen {
  Boolean visitado;   /* inicialmente falso */
  ListaGen prox;
  Boolean eAtomo;
  union {
    int atomo;       /* caso de 'eAtomo' verdadeiro */
    ListaGen lista;  /* caso de 'eAtomo' falso */
  } info;
} RegListaGen;
```


Contagem geral de ocorrências de átomos

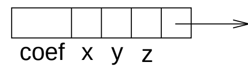
```
int ContaAtomos(ListaGen p) {
  int s = 0;
  while ((p!=NULL) && !(p->visitado)) {
      p->visitado = true;
      if (p->eAtomo)
        s++;
      else
        s += ContaAtomos(p->info.lista);
    }
    p = p->prox;
  }
  return s;
} /* ContaAtomos */
```

Problema: restauração dos valores do campo *visitado* para próximo percurso.

Exemplo de aplicação: polinômios em múltiplas variáveis

$$P(x,y,z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z - 6x^3y^4z + 2yz$$

Representação possível para cada termo:

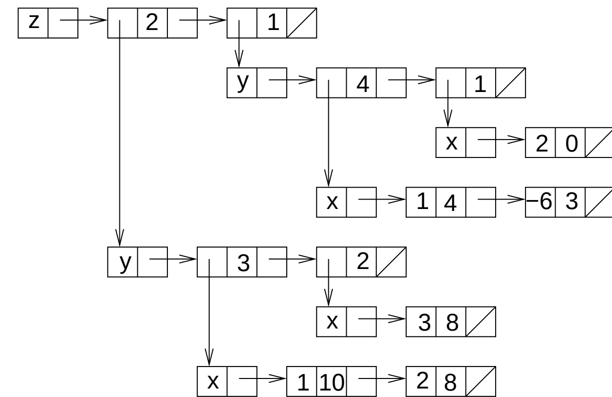


Problema: muito inflexível, somente para polinômios em três variáveis.

Alternativa: um polinômio em $k \geq 1$ variáveis pode ser considerado um polinômio em uma variável, com coeficientes que são polinômios em $k-1$ variáveis, etc:

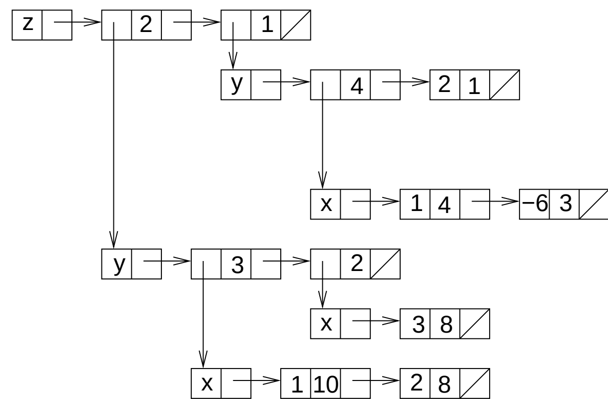
$$P(x,y,z) = ((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 - 6x^3)y^4 + 2x^0y)z$$

Representação de polinômio (alternativa 1)



Representação uniforme em todos os níveis.

Representação de polinômio (alternativa 2)



Representação que elimina polinômios "degenerados".

Declaração de tipo

```
typedef struct Termo *ApTermo;  typedef ApTermo Polinomio;
typedef struct Termo {
  Polinomio prox;
  Boolean eCabeca;
  union {
    char variavel;        /* se é cabeça */
    struct {              /* se é termo */
      int expoente;
      Boolean coefInteiro;
      union {
        int coefInt;
        Polinomio coefPolin;
      } coef;
    } termo;
  } no;
} Termo;
```

Exercício

Escrever as funções de soma e multiplicação para polinômios em múltiplas variáveis.

Compressão de textos: codificação de Huffman

Compressão de textos

- representação normal: um byte (8 bits) por caractere (alfabetos “comuns”)
- compressão por contagem (*run-length encoding*)
- codificação de Huffman
- algoritmos de codificação aritmética (Lempel-Ziv – zip, gzip, winzip, ...)

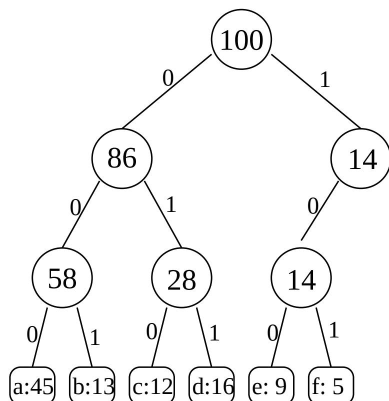
Codificação de Huffman

- explora freqüências de ocorrência de caracteres
- exemplo de alfabeto: $\mathcal{A} = \{a, b, c, d, e, f\}$

	a	b	c	d	e	f
Freqüência de cada letra	45	13	12	16	9	5
Codificação usando 3 bits	000	001	010	011	100	101
Codificação de tamanho variável	0	101	100	111	1101	1100

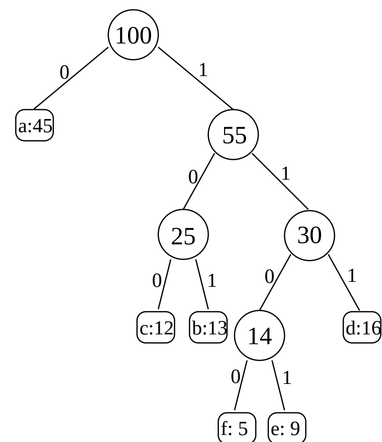
- para um arquivo de 100.000 caracteres:
  - codificação fixa: 300.000 bits
  - codificação variável: 224.000 bits (economia de 25%)

Árvores binárias de codificação: código fixo



$abc = 000\|001\|010 = 000001010.$

Árvores binárias de codificação: código variável



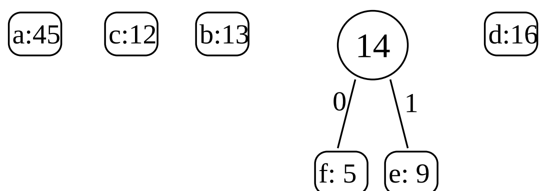
$abc = 0\|101\|100 = 0101100$ (código de uma letra não pode constituir o prefixo de uma outra letra).

Construção da árvore de Huffman

Situação inicial: uma floresta de folhas

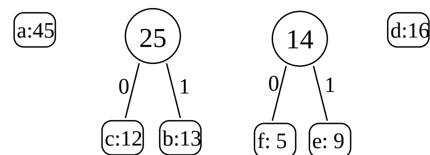


Passo 1:

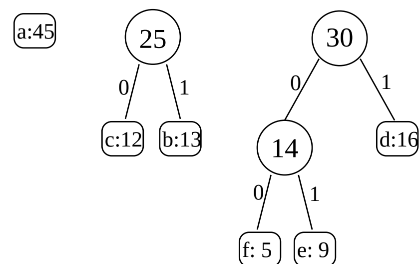


Construção da árvore de Huffman (cont.)

Passo 2:

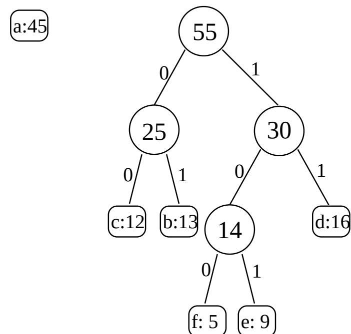


Passo 3:

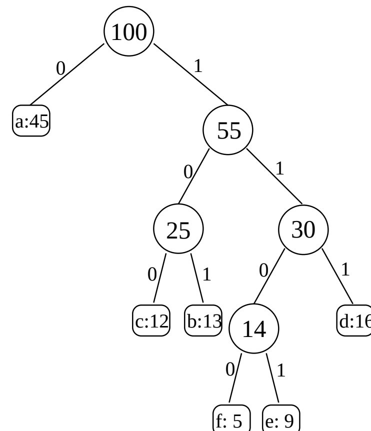


Construção da árvore de Huffman (cont.)

Passo 4:



Passo 5:



Algoritmo de construção da árvore (*algoritmo guloso*)

1. Construa uma floresta de folhas, cada uma correspondendo a um caractere, com a respectiva freqüência como seu peso.
2. Enquanto a floresta tiver mais de uma árvore, repita:
   - procure na floresta duas árvores t_1 e t_2 de peso mínimo
   - construa uma nova árvore t, com subárvores t_1 e t_2, e com peso que é a soma dos pesos das duas subárvores
   - remova t_1 e t_2 da floresta, e insira t.

Obs.: A escolha de árvores de peso mínimo pode ser implementada por uma fila de prioridades.

Observações sobre a codificação de Huffman

- adotadas certas hipóteses, demonstra-se a otimalidade de compressão
- algoritmo de compressão: para cada letra, deve acessar a folha correspondente da árvore e reconstruir o caminho à raiz – pode ser preprocessado
- descompressão: percorre a árvore a partir da raiz seguindo o caminho indicado por *bits* da codificação
- variantes:
  - árvore fixa, por exemplo, uma para cada língua
  - árvore por texto (acompanha o arquivo)
  - árvores dinâmicas (Faller, Gallager e Knuth).

Gerenciamento de memória

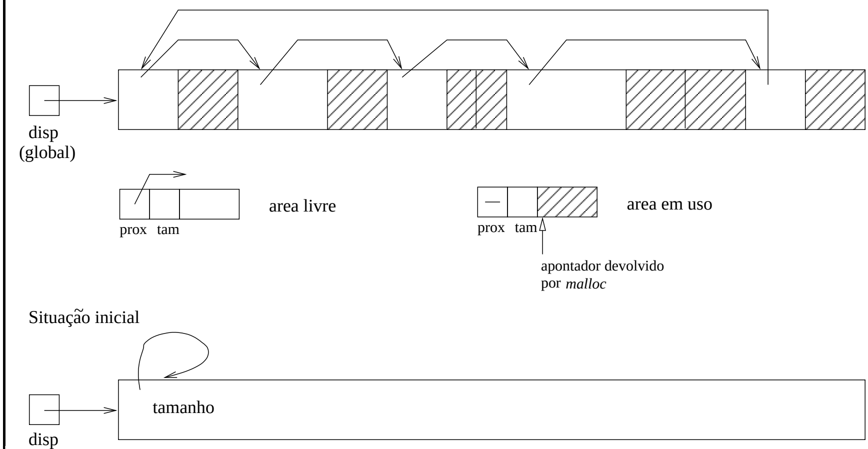
Gerenciamento de memória

Vários aspectos:

- registros de tamanho fixo ou variável
- gerenciamento explícito (*malloc* e *free*) ou implícito (coleta de lixo e contagem de referências)

Gerenciamento explícito de memória com listas ligadas

(ordem crescente dos apontadores)



Gerenciamento explícito

Esboço de uma versão muito simples de *malloc(n)*

Seja f o tamanho da parte fixa de cada área (apontador mais campo de tamanho):

1. procure na lista *disp* o primeiro elemento (ou o elemento de tamanho mínimo) p com *tamanho*$\geq n + f$
2. remova p da lista *disp*
3. quebre a área apontada por p em duas partes: uma p_1 de tamanho $n + f$ e outra p_2 com o que sobrar (se houver sobra suficiente)
4. insira p_2 na lista *disp* (se existir)
5. devolva $p_1 + f$

Gerenciamento explícito:

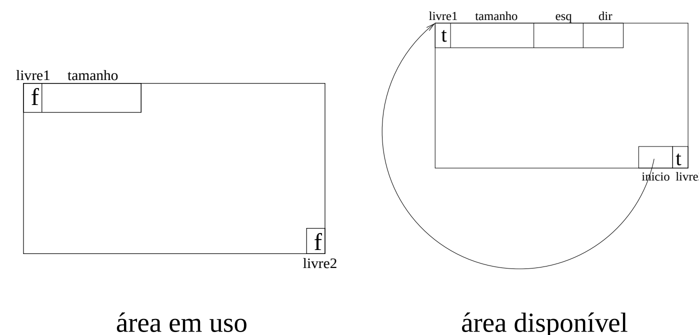
Esboço de uma versão muito simples de *free(p)*

1. procure na lista *disp* o ponto de inserção para p (ordem crescente dos apontadores)
2. verifique se o predecessor e/ou sucessor de p neste ponto são adjacentes à área apontada por p
3. se for possível, junte a área liberada à predecessora e/ou sucessora, modificando os campos de *tamanho*
4. atualize a lista

Gerenciamento explícito: Problemas:

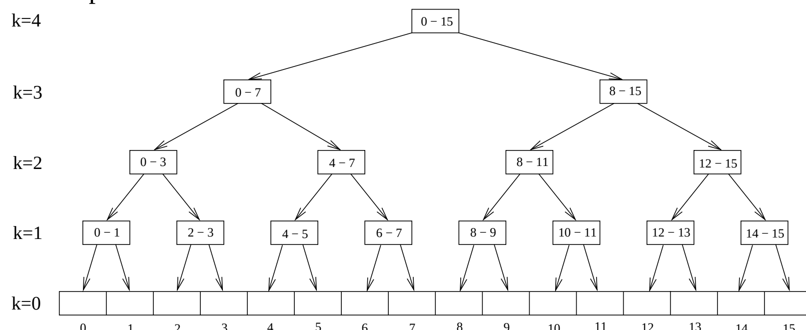
- o que fazer quando *malloc* não acha uma área de tamano suficiente – requer outra área ao sistema operacional
- fragmentação – após várias alocações e liberações, com tamanhos variáveis, haverá tendência a produzir muitas área livres pequenas
- busca numa lista ligada pode ser ineficiente:
  - blocos com *marcas de fronteira*
  - sistema de *blocos conjugados*

Marcas de fronteira (*boundary tags*)



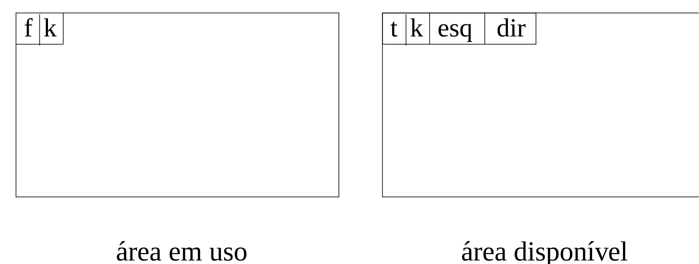
Sistema de blocos conjugados (*buddy system*)

Árvore para $m = 4$:



- cada nó representa um bloco de alocação.
- cada folha representa um bloco mínimo de alocação.
- áreas conjugadas (irmãos) facilmente reconhecidos pelos índices sob forma binária

Sistema de blocos conjugados: Áreas



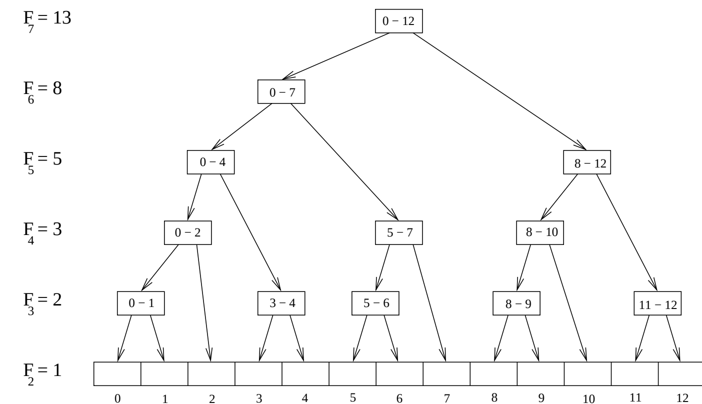
- listas disponíveis duplamente ligadas, uma para cada valor de k
- tamanho de cada área: 2^k unidades mínimas

Sistema de blocos conjugados: Esboço da função *malloc(n)*

Seja f o tamanho da parte fixa de cada bloco (marca de uso, tamanho e os apontadores):

1. procura um k tal que $2^k \geq n + f$, e a lista de blocos para k não está vazia; remova desta lista um bloco p
2. se $2^{k-1} \geq n + f$, quebra o bloco p em dois (conjugados), acerta os tamanhos e insere um deles na lista de blocos para tamanho $k - 1$
3. repete o passo anterior para $k - 2$, $k - 3$, ..., enquanto possível
4. devolve o apontador $p + f$

Sistema de blocos conjugados: alternativa com números de Fibonacci



Gerenciamento implícito (coleta de lixo – *garbage collection*)

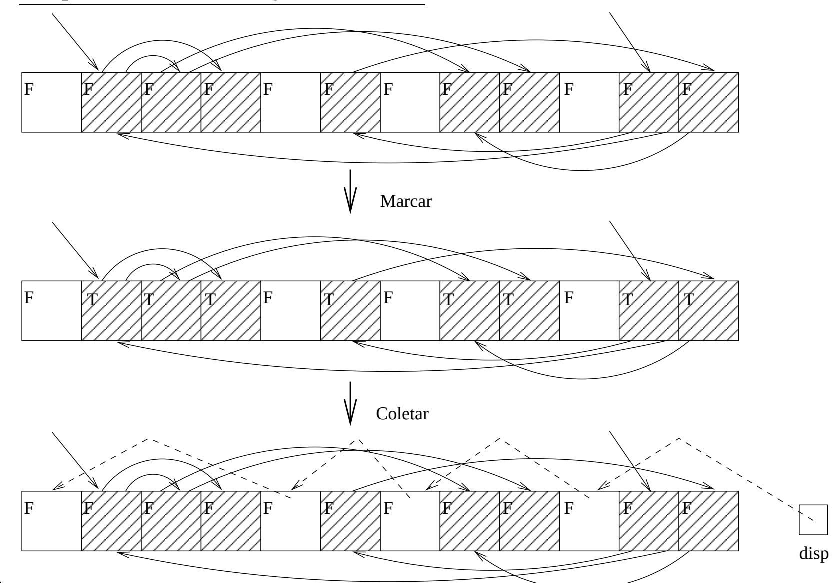
Características:

- operação de alocação implícita em algumas operações
- não existe operação de liberação explícita (em geral)
- liberação de memória realizada, em geral, quando não há mais espaço para alocar
- exemplos: Java, LISP, Prolog, Perl, Python, Modula-3, ...
- contra-exemplos: C, Pascal, C++, ...

Fases:

- marcação
- coleta ou compactação
- caso haja compactação: cálculo de destino; atualização dos apontadores; cópia dos blocos

Esquema de marcação e coleta



Coleta de lixo versão simplificada – declarações

Hipóteses simplificadoras:

- tamanho fixo de registros
- localização conhecida dos apontadores

```
typedef struct Bloco *ApBloco;
typedef struct Bloco {
    Boolean marca;
    ApBloco destino;    /* se houver compactação */
    . . .
    int numAps;
    ApBloco aponts[NUM_MAX_APONTS];
} Bloco;

ApBloco disp;          /* inicialmente todos os blocos */
Bloco memoria[TAM_MEM_DIN];
```

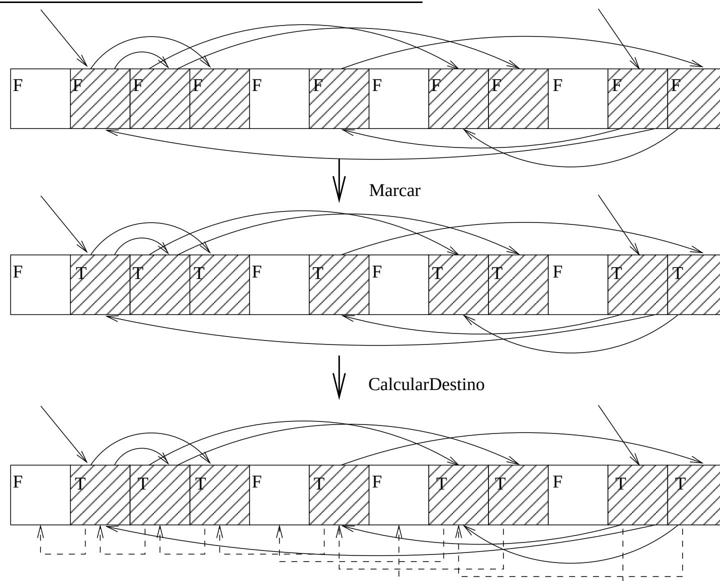

Coleta de lixo versão simplificada de marcação e coleta

```
void Marcar(ApBloco p) {
  int i;
  if (p!=NULL) {
    if (!p−>marca) {
      p−>marca = true;
      for (i=0; i<p−>numAps; i++)
        Marcar(p−>aponts[i]);
    }
  }
} / * Marcar */
```

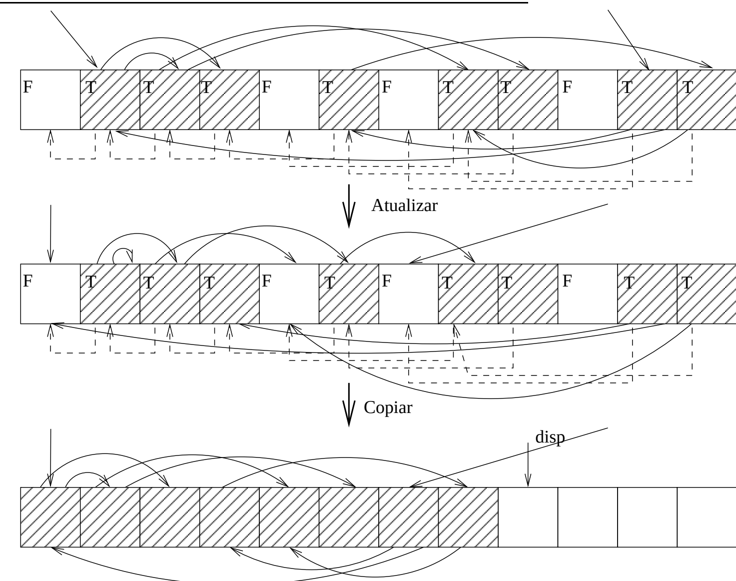
```
void Coleta() {
  int i;
  disp = NULL;
  for (i=0; i<TAM_MEM_DIN; i++)
    if (memoria[i].marca) /* em uso */
        memoria[i].marca = false;
      else { /* insere na lista disponível */
        memoria[i].aponts[0] = disp;
        disp = &(memoria[i]);
      }
} /* Coleta */
```

A função *Marca* deve ser chamada para cada variável apontadora na pilha de execução.

Esquema marcação e compactação



Esquema marcação e compactação (cont.)



Coleta de lixo com compactação versão simplificada de cálculo de destino

```
void CalcularDestino() {
   int i,j = 0;
   for (i=0; i<TAM_MEM_DIN; i++)
      if (memoria[i].marca) {
         memoria[i].destino = &(memoria[j]);
         j++;
   }
   disp = &(memoria[j]); /* primeiro bloco livre */
} /* CalcularDestino */
```

Como os blocos disponíveis ficarão consecutivos, basta colocar em *disp* o apontador ao primeiro, sem formar uma lista ligada.

Coleta de lixo com compactação versão simplificada de atualização de apontadores

```
void Atualiza() {
   int i,j = 0;
   for (i=0; i<TAM_MEM_DIN; i++)
      if (memoria[i].marca)
       for (j=0; j<memoria[i].numAps; j++) {
          memoria[i].aponts[j] = (memoria[i].aponts[j])->destino;
       }
} /* Atualiza */
```

Devem ser atualizadas também todas as variáveis apontadoras na pilha de execução.

Coleta de lixo com compactação versão simplificada de cópia

```
void Move() {
   int i;
   for (i=0; i<TAM_MEM_DIN; i++)
      if (memoria[i].marca) {
         memoria[i].marca = false;
         *(memoria[i].destino) = memoria[i];
      }
} /* Move */
```

Adaptação para blocos de tamanho variável: introduzir campo *tamanho* em cada bloco.

Contagem de referências

Supõe que cada bloco alocado possui um campo inteiro *refs* contendo o número de variáveis (normais ou dinâmicas) que apontam para o bloco. O tipo do bloco e a implementação do comando da forma $p = q$ são:

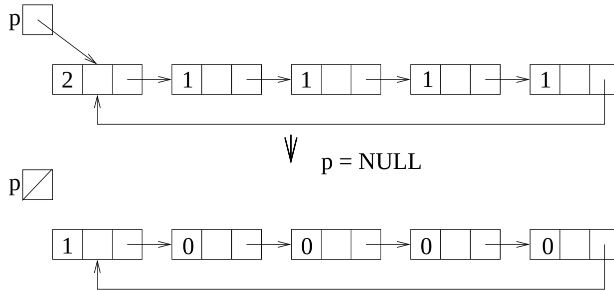
```
typedef struct Bloco *ApBloco;
typedef struct Bloco {
   int refs;
   . . .
   int numAps;
   ApBloco aponts[NUM_MAX_APONTS];
} Bloco;
```

```
if (q!=NULL) (q->refs)++;
if (p!=NULL) {
  (p->refs)--;
  if ((p->refs)==0)
    DesalocaRefs(p);
}
p = q;
```

Supõe que as variáveis apontadoras são inicializadas com *NULL*.

Contagem de referências: problemas

- dependendo das circunstâncias, o tempo de execução de comando de atribuição entre apontadores é imprevisível (a função *DesalocaRefs* é recursiva)
- o método, como exposto, não funciona para estruturas com circularidades:



Tipos de Dados Abstratos e Objetos

TADs em C: exemplo 1

Arquivo figuras.h:

```
typedef void * Figura;
Figura Retangulo(float alt, float larg);
Figura Circulo(float raio);
Figura Quadrado(float lado);
float Area(Figura fig);
void Transladar(Figura fig, float dx, float dy);
void Desenhar(Figura fig);
```


TADs em C: exemplo 1

Arquivo main.c:

```
#include "figuras.h"
int main() {
  Figura c = Circulo(10.0);
  Figura r = Retangulo(10.0, 20.0);
  Figura q = Quadrado(50.0);
  Transladar(r,5.0,8.0);
  Desenhar(q);
  printf("%f\n",Area(c));
  printf("%f\n",Area(r));
  printf("%f\n",Area(q));
  return 0;
} /* main */
```

TADs em C: exemplo 1 (cont.)

Arquivo figuras.c: declarações

```
typedef enum { RETANGULO, CIRCULO, QUADRADO } FormaFigura;

typedef struct {
 FormaFigura forma;
 float posx, posy;
 union {
  struct { float alt, larg; } lados;
  float raio;
 } dados;
} _RegFigura, * _Figura;
```


TADs em C: exemplo 1 (cont.)

Arquivo figuras.c: construtores

```
Figura Retangulo(float alt, float larg) {
 _Figura f = malloc(sizeof(_RegFigura));
 f->forma = RETANGULO;
 f->posx = 0.0;
 f->posy = 0.0;
 f->dados.lados.alt = alt;
 f->dados.lados.larg = larg;
 return f;
} /* Retangulo */
```


Arquivo figuras.c: construtores (cont)

```
Figura Circulo(float raio) {
 _Figura f = malloc(sizeof(_RegFigura));
 f->forma = CIRCULO;
 f->posx = 0.0;
 f->posy = 0.0;
 f->dados.raio = raio;
 return f;
} /* Circulo */

Figura Quadrado(float lado) {
 _Figura f = Retangulo(lado,lado);
 f->forma = QUADRADO;
 return f;
} /* Quadrado */
```


Arquivo figuras.c: funções

```
float Area(Figura fig) {
 _Figura f = fig;
 switch (f->forma) {
  case RETANGULO:
  case QUADRADO:
   return (f->dados.lados.alt)*(f->dados.lados.larg);
  case CIRCULO:
   return PI*(f->dados.raio)*(f->dados.raio);
  default:
   exit(1); /* Impossível */
 }
} /* Area */
```

Arquivo figuras.c: funções (cont.)

```
void Transladar(Figura fig, float dx, float dy) {
  _Figura f = fig;
  f->posx += dx;
  f->posy += dy;
} /* Transladar */


void Desenhar(Figura f) {
  /* Não foi implementada */
} /* Desenhar */
```

(Fim do exemplo 1 de TAD.)

TADs em C: exemplo 2

Arquivo figuras.h:

```
typedef void * Figura;
Figura Retangulo(float alt, float larg);
Figura Circulo(float raio);
Figura Quadrado(float lado);
float Area(Figura fig);
void Transladar(Figura fig, float dx, float dy);
void Desenhar(Figura fig);
```

(Idêntico ao exemplo 1; `main.c` também é.)

TADs em C: exemplo 2 (cont.)

Arquivo figuras.c: declarações
(somente *Area* foi transformado em verdadeiro método)

```
typedef float funcArea(Figura);  /* tipo função */
typedef enum { RETANGULO, CIRCULO, QUADRADO } FormaFigura;

typedef struct {
  FormaFigura forma;
  float posx, posy;
  funcArea *Area;   /* verdadeiro método */
  union {
    struct {float alt, larg; } lados;
    float raio;
  } dados;
} _RegFigura, * _Figura;
```


TADs em C: exemplo 2 (cont.)

Arquivo figuras.c: funções do tipo *funcArea*

```
float AreaRetangulo(Figura fig) {
  _Figura f = fig;
  return (f->dados.lados.alt)*(f->dados.lados.larg);
} /* AreaRetangulo */

float AreaCirculo(Figura fig) {
  _Figura f = fig;
  return PI*(f->dados.raio)*(f->dados.raio);
} /* AreaCirculo */
```

TADs em C: exemplo 2 (cont.)

Arquivo figuras.c: construtores

```
Figura Retangulo(float alt, float larg) {
  _Figura f = malloc(sizeof(_RegFigura));
  f−>forma = RETANGULO;
  f−>posx = 0.0;
  f−>posy = 0.0;
  f−>Area = AreaRetangulo;  /* método */
  f−>dados.lados.alt = alt;
  f−>dados.lados.larg = larg;
  return f;
} /* Retangulo */
```


Arquivo figuras.c: construtores (cont)

```
Figura Circulo(float raio) {
  _Figura f = malloc(sizeof(_RegFigura));
  f−>forma = CIRCULO;
  f−>posx = 0.0;
  f−>posy = 0.0;
  f−>Area = AreaCirculo;  /* método */
  f−>dados.raio = raio;
  return f;
} /* Circulo */

Figura Quadrado(float lado) {
  _Figura f = Retangulo(lado,lado);
  f−>forma = QUADRADO;
  return f;
} /* Quadrado */
```


Arquivo figuras.c: funções

```
float Area(Figura fig) {
  return ((_Figura)fig)−>Area(fig);
}

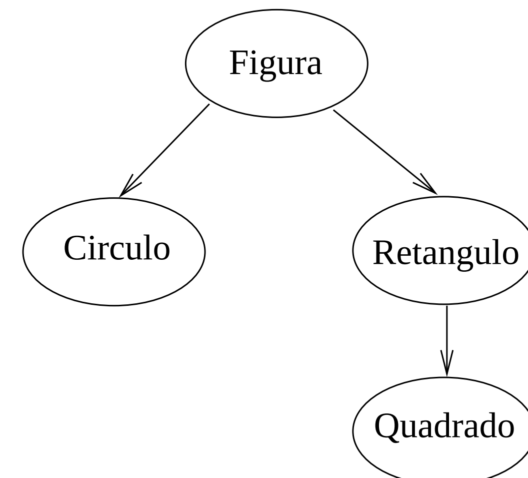
void Transladar(Figura fig, float dx, float dy) {
  _Figura f = fig;
  f−>posx += dx;
  f−>posy += dy;
} /* Transladar */


void Desenhar(Figura fig) {
  /* Não foi implementada */
} /* Desenhar */
```

(Fim do exemplo 2 de TAD.)

Objetos: classes de figuras geométricas

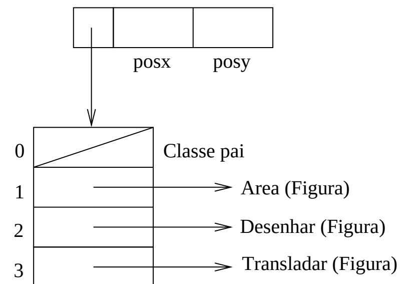
Hierarquia das classes:



Omitiremos vários aspectos como por exemplo polimorfismo, visibilidade etc.

Classe *Figura* (usando uma linguagem fictícia)

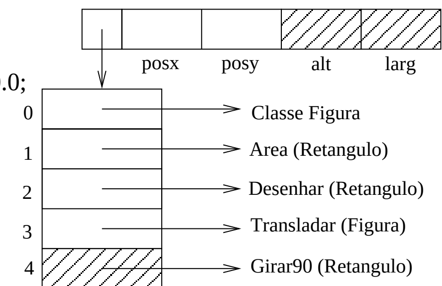
```
class Figura {
   float posx, posy;
   /* não existe construtor */
   float Area() { };
   void Desenhar() { };
   float Transladar(float dx, dy) {
      this.posx += dx;
      this.posy += dy;
   }
} /* Figura */
```



Todos os objetos de uma classe apontam para a mesma tabela de métodos. Pode haver mais informações. Neste exemplo, todas as funções foram transformadas em métodos.

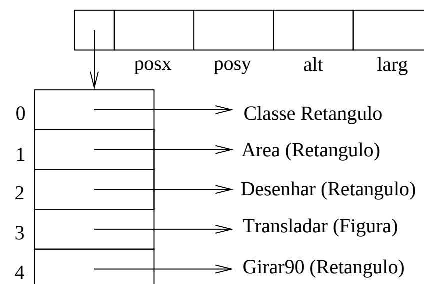
Classe *Retangulo*

```
class Retangulo extends Figura {
   float alt, larg;
   Retangulo(float a, float l) {
      this.alt = a;    this.larg = l;
      this.posx = 0.0;  this.posy = 0.0;
   }
   float Area() {
      return alt*larg;
   }
   void Desenhar() { . . . };
   Retangulo Girar90() {
    return new Retangulo(this.larg,this.alt);
   }
} /* Retangulo */
```



Classe *Quadrado*

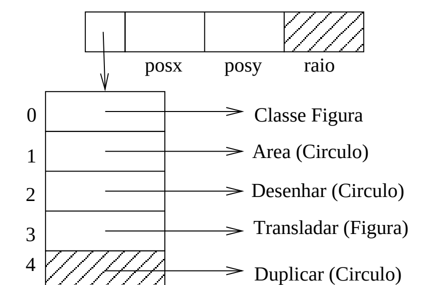
```
class Quadrado extends Retangulo {
   Quadrado(float l) {
      super(l,l);
   }
} /* Retangulo */
```



Somente o construtor é diferente da classe *Retangulo*.

Classe *Circulo*

```
class Circulo extends Figura {
   float raio;
   Circulo(float r) {
      this.posx = 0.0;
      this.posy = 0.0;
      this.raio = r;
   }
   float Area() {
      return PI*sqr(raio);
   }
   void Desenhar() { . . . };
   void Duplicar() {
     this.raio = 2.0*this.raio;
   }
} /* Retangulo */
```



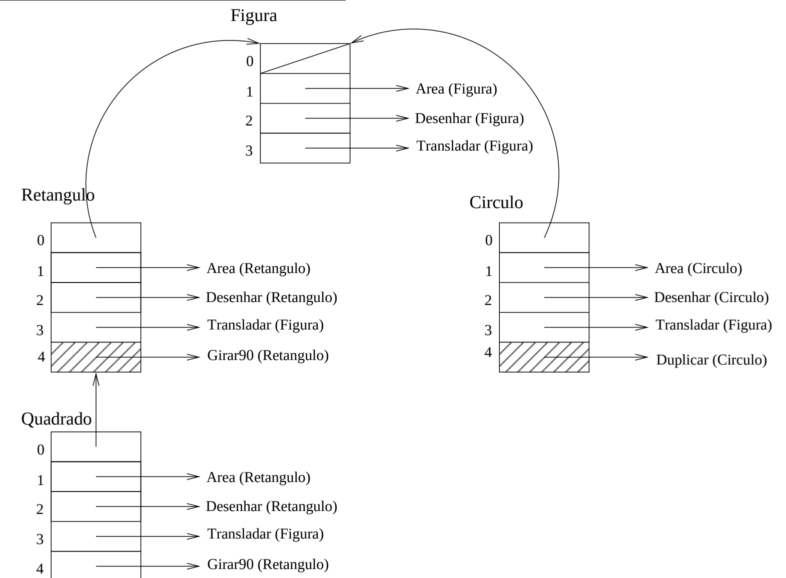
Exemplo de uso

```
int main() {
  Figura f = new Circulo(10.0);
  Retangulo r = new Retangulo(10.0,20.0);
  Quadrado q = new Quadrado(50.0);
  Circulo c = new Circulo(30.0);
  printf("%f\n",f.Area());
  printf("%f\n",r.Area());
  c.Desenhar();
  f = q;
  f.Transladar(5.0,8.0);
  f.Desenhar();
  printf("%f\n",f.Area());
```

```
/* comandos inválidos */
  f.Duplicar();
  f.Girar90();
  c.Girar90();
  c = r;
  q = f;
} /* main */
```

Os comandos inválidos seriam detectados pelo compilador.

Descrição final das classes



Algoritmos de ordenação

Generalidades

- ordenação interna e externa
- ordenação ótima por comparações: $\mathrm{O}(n \log n)$
- exemplos de algoritmos por comparação e sua eficiência:
  - transposição (*bubblesort*, *quicksort*)
  - inserção (inserção simples, *shellsort*)
  - seleção (seleção simples, *heapsort*)
  - intercalação (iterativo, recursivo)
- outros algoritmos: distribuição (*radix sort*)

Declarações comuns

```
typedef struct Vetor {
  int n;            /* tamanho */
  int dados[1];  /* para o compilador */
} Vetor, *ApVetor;
```

```
void troca(int *x, int *y) {
  int t = *x;
  *x = *y;
  *y = t;
} /* troca */
```


Algoritmo "bubble sort": sempre O(n^2)

```
void bubbleSort(ApVetor v) {
 /*Exemplo de transposição */
 int n = v->n;
 int *d = (v->dados);
 int i,j;
 for (i=n-1; i>0; i--)
  for (j=0; j<i; j++)
   if (d[j]>d[j+1])
    troca(&d[j],&d[j+1]);
} /* bubbleSort */
```


Inserção simples: pior caso - O(n^2), melhor caso O(n); bom para dados parcialmente ordenados

```
void insercao(ApVetor v) {
 int n = v->n;
 int *d = (v->dados);
 int i,j;
 int t;
 for (i=0; i<n-1; i++) {
  t = d[i+1];
  j = i;
  while ((j>=0)&&(t<d[j])) {
   d[j+1] = d[j];
   j--;
  }
  d[j+1] = t;
 }
} /* insercao */
```


Seleção simples: sempre - O(n^2)

```
void selecao(ApVetor v) {
 int n = v->n;
 int *d = (v->dados);
 int i,j;
 int p;
 for (i=0; i<n-1; i++) {
  p = i;
  for (j=i+1; j<n; j++)
   if (d[j]<d[p])
    p = j;
  troca(&d[i],&d[p]);
 }
} /* selecao */
```

Algoritmo “quick sort”:

```
void quickSort(ApVetor v) {

  quickSortAux(v,0,(v−>n)−1);

} /* quickSort */
```

Eficiência:

- pior caso – $\mathrm{O}(n^2)$
- média, melhor caso – $\mathrm{O}(n \log n)$
- na prática, quase sempre $\mathrm{O}(n \log n)$


```
void quickSortAux(ApVetor v, int esq, int dir) {
 /* supõe esq<=dir */
 int *d = (v−>dados);
 int i = esq, j = dir;
 int pivot = d[(int)((esq+dir)/2)];
 do {
  while (d[i]<pivot) i++;
  while (d[j]>pivot) j−−;
  if (i<=j) {
   troca(&d[i],&d[j]);
   i++; j−−;
  }
  } while (i<=j);
 if (esq<j) quickSortAux(v,esq,j);
 if (dir>i) quickSortAux(v,i,dir);
} /* quickSortAux */
```


Intercalação iterativa $\mathrm{O}(n \log n)$

```
void intercalaIterativo(ApVetor v) {
 /* Ordena de 2 em 2, de 4 em 4, ..., usando intercalação */
 int n = v−>n;
 int td = 1;              /* 1, 2, 4, ... */
 int esq, dir, ld;
 int tamanho = sizeof(Vetor)+sizeof(int)*(n−1);
 Boolean par = false;
 ApVetor w = (ApVetor)malloc(tamanho);
 w−>n = v−>n;
```

(continua)


```
 while (td<n) {
  esq = 0;  par = !par;
  do {
   dir = esq+td; ld = dir+td;
   if (dir>=n) {  /* lado direito vazio */
    dir = n; ld = n−1;
   } else if (ld>n)
    ld = n;
   if (par) intercalaIterativoAux(v,w,esq,dir,ld);
   else  intercalaIterativoAux(w,v,esq,dir,ld);
   esq = dir+td;
  } while (esq<n);
  td = 2*td;
 }
 if (par)  memcpy(v,w,tamanho);
 free(w);
} /* intercalaIterativo */
```

```
void intercalaIterativoAux(ApVetor v, ApVetor w, int esq, int dir, int ld) {
 /* Intercala v.dados[esq:dir-1] e v.dados[dir:ld-1] em w.dados[esq:ld-1] */
 int *dv = (v->dados), *dw = (w->dados);
 int i = esq, j = dir, k= esq;
 while ((i<dir)&&(j<ld)) {
  if (dv[i]<=dv[j]) {
   dw[k] = dv[i]; i++;
  } else {
   dw[k] = dv[j]; j++;
  }
  k++;
 }
 while (i<dir) { dw[k] = dv[i]; i++; k++; }
 while (j<ld) { dw[k] = dv[j]; j++; k++; }
} /* intercalaIterativoAux */
```


Intercalação recursiva $O(n \log n)$

```
void intercalaRecursivo(ApVetor v) {
 int n = v->n;
 if (n>1) {
  int *dv = v->dados;  ApVetor v1,v2;
  int i, nv1 = (int)(n/2), nv2 = n-nv1;
  v1 = (ApVetor)malloc(sizeof(Vetor)+sizeof(int)*(nv1-1));
  v2 = (ApVetor)malloc(sizeof(Vetor)+sizeof(int)*(nv2-1));
  v1->n = nv1;  v2->n = nv2;
  for (i=0; i<nv1; i++)  (v1->dados)[i] = dv[i];
  for (i=0; i<nv2; i++)  (v2->dados)[i] = dv[i+nv1];
  intercalaRecursivo(v1); intercalaRecursivo(v2);
  intercalaRecursivoAux(v1,v2,v);
  free(v1);  free(v2);
 }
} /* intercalaRecursivo */
```



```
void intercalaRecursivoAux(ApVetor u,  ApVetor v, ApVetor w) {
 /* Intercala os vetores u e v, deixando o resultado em w. */
 int i = 0, j = 0, k;
 int nu = u->n,  nv = v->n,  n = nu+nv;
 int *du = (u->dados),  *dv = (v->dados),  *dw = (w->dados);
 for (k=0; k<n; k++) {
  if ((i<nu)&&(j<nv)) {
   if (du[i]<=dv[j]) { dw[k] = du[i]; i++; }
   else { dw[k] = dv[j]; j++; }
  } else {
   if (i<nu) { dw[k] = du[i]; i++; }
   else { dw[k] = dv[j]; j++; }
  }
 }
} /* intercalaRecursivoAux */
```


Fim!

